Python

Escreva seus primeiros programas

FELIPE CRUZ

Prefácio

Python é uma linguagem de programação que vem sendo empregada na construção de soluções para os mais diversos fins educacionais, comerciais e científicos e plataformas web, desktop e, mais recentemente, móvel. É uma linguagem de fácil aprendizado, expressiva, concisa e muito produtiva; por isso, sua adoção tem crescido bastante nos últimos anos pelos mais variados perfis de profissionais no meio científico e acadêmico, tanto para desenvolvimento de ferramentas quanto para ensino de algoritmos e introdução à programação.

É uma linguagem de uso gratuito e de código-fonte aberto, compatível com os principais sistemas operacionais, como: Linux, OSX, Windows, BSDs etc. Ela conta com uma vasta biblioteca padrão e documentação que possibilitam que muitas coisas sejam feitas sem dependências adicionais.

Apesar de ser simples de aprender, Python é uma linguagem bastante poderosa e flexível. Essa combinação resulta em um rico ecossistema que melhora a produtividade dos desenvolvedores. Isso tudo torna a decisão de aprender Python importantíssima, pois muitos horizontes abrem-se quando se domina uma linguagem com ecossistema tão poderoso quanto o dela.

Se você está lendo este livro, possivelmente se interessou por Python em algum momento. Então, espero que ele possa contribuir, de alguma forma, com o seu aprendizado. O livro apresenta a linguagem de uma forma contextualizada e, por isso, ao longo dele, vamos criar um aplicativo para analisar dados públicos do Governo Federal. A ideia é apresentar motivações práticas, para depois demonstrar os recursos da linguagem que nos permitem tratar a motivação inicial.

Este livro está dividido em duas partes. A primeira foca nos aspectos fun-

damentais de Python, muitos dos quais já são conhecidos por quem já tem experiência com outras linguagens de programação. Já na segunda, passaremos a olhar características mais específicas da linguagem que, embora possam não ser exclusivas, são marcantes e devem ser tratadas com mais cuidado.

Por que Python 3?

O ano de 2014 foi um ano chave na adoção de Python3, por grande parte da comunidade. Muitos projetos relevantes foram portados ou lançaram versões compatíveis nesse ano. A família 3.4 tem maior aceitação que as anteriores da versão 3 e, inclusive, foi adotada como versão padrão em alguns sistemas operacionais. Python 3 nunca chamou tanta atenção como agora, então nada mais justo que um livro o tenha como assunto base.

Nosso objetivo é apresentar os recursos básicos da linguagem, ideias e conceitos centrais construindo um aplicativo simples de leitura e manipulação de dados.

O livro é conceitual e prático ao mesmo tempo, para que o aprendizado seja mais profundo sem ser chato e difícil de entender. Livros extremamente práticos podem, muitas vezes, pular conceitos e ideias centrais do assunto abordado, ao ponto que textos apenas conceituais podem ficar cansativos e teóricos demais.

Assim, o propósito é fazer com que você entenda melhor o universo Python, ao mesmo tempo em que aprende a usar na prática os principais recursos da linguagem.

Público-alvo

Este é um livro para iniciantes em programação ou desenvolvedores avançados com pouca experiência em Python. Se você for um iniciante, leia-o com calma para não acumular dúvidas ao longo do aprendizado. Em termos de complexidade, a maioria dos exemplos é bem simples e foca em passar para o leitor como usar os recursos disponíveis da melhor forma. Em termos conceituais, todas as explicações buscam ser completas para não exigir consulta a fontes externas.

Caso você já tenha alguma experiência em programação, mas não conheça Python, o livro contribui para que você rapidamente descubra como

implementar nessa linguagem coisas que são comuns em outras. Além disso, os capítulos abordam aspectos bem específicos, com explicações conceituais e exemplos práticos.

Aqui explico como as coisas funcionam no universo Python e apresento um pouco da visão *pythônica* para o leitor. Até mesmo as partes mais triviais podem conter *insights* importantes sobre o comportamento da linguagem ou decisões de design adotadas.

Os pedaços de código apresentados serão autocontidos e permitirão ao leitor modificar e obter resultados diferentes. Todos os códigos são explicados, muitas vezes linha a linha, para que você entenda claramente o objetivo e a função de cada parte.

Caso tenha alguma dúvida ou sugestão, procure a comunidade do livro para tirar dúvidas. Ela está disponível em livro-python3@googlegroups.com. Você será muito bem-vindo!

Os códigos-fonte dos exemplos utilizados ao longo do livro podem ser encontrados em: https://github.com/felipecruz/exemplos

Sobre o autor

Felipe Cruz é desenvolvedor de software há 10 anos. Trabalhou no mercado corporativo e de startups e recentemente atua como Cientista de Dados, área onde reencontrou sua paixão pela construção de software. Mestrando em Computação pela PUC-Rio, também estuda Aprendizado de Máquinas e atualmente busca novas formas de resolver o problema da recomendação e o aprendizado não supervisionado de atributos.

Pythonista há 6 anos, palestrante e participante de diversas Python Brasil, sempre buscou compartilhar conhecimento e aprender cada vez mais sobre a linguagem e o ecossistema ao longo desses anos.

Agradecimentos

Agradeço, primeiramente, à minha família pelo apoio e paciência ao longo do tempo de criação do livro. Em especial, para minhas filhas Maya e Liz, minha esposa Fernanda, mãe e irmãs Ilka, Ilka e Ieda.

Também agradeço muito pela confiança e paciência da equipe Casa do Código ao longo desta árdua jornada. Em diversos momentos, não consegui produzir o que era esperado mas mesmo assim sempre pude contar com a paciência e apoio de todos.

Por fim, agradeço a todos que trabalharam comigo, em especial para os diversos amigos que fiz no universo Python.

Agradecimentos

Erratas

Esse capítulo é dedicado aos contribuidores do livro:

- Sandro Dutra - Errata - Seção 2.2
- Francisco André - Errata - Seção 3.1
- André França - Errata - Seção 4.1

Sumário

Versão: 18.9.26

Parte I

Seus primeiros passos com o Python

Ao longo do livro, vamos desenvolver um aplicativo que realiza um processo relativamente complexo para conseguir extrair informação dos dados públicos do Governo Federal. Basicamente, ele faz um download da internet, abre o arquivo trazido e faz algumas consultas no conteúdo. Para que diversas buscas possam ser feitas, ele também cria uma abstração em cima dos dados, como se fosse um pequeno banco de dados. Por meio de toda a motivação prática para os aspectos apresentados nesse aplicativo, demonstraremos diversos recursos da linguagem Python: criação de funções, uso de loops, tratamento de erro, criação de classes e outros.

Nesta primeira parte, vamos abordar os aspectos mais básicos de Python, que geralmente existem em outras linguagens, e mostrar que não existem limites no resultado ou no impacto dos nossos programas, mesmo com o básico.

E aí? Está preparado para começar?

Capítulo 1

Iniciando com Python

1.1 A linguagem Python

Python é uma linguagem interpretada de alto nível e que suporta múltiplos paradigmas de programação: imperativo, orientado a objetos e funcional. É uma linguagem com tipagem dinâmica e forte, escopo léxico e gerenciamento automático de memória. Possui algumas estruturas de dados embutidas na sintaxe – como tuplas, listas e dicionários – que aumentam muito a expressividade do código. Além de tudo isso, Python possui *baterias inclusas*, uma expressão que se refere a uma vasta biblioteca padrão com diversos utilitários poderosos.

A sintaxe básica de Python é bem simples e pode ser aprendida rapidamente. Com mais prática, elementos mais complexos – como *comprehensions*, *lambdas*, *packing* e *unpacking* de argumentos – vão passando a fazer parte do dia a dia do programador. Esta linguagem tem uma característica não muito

comum, que é o uso da indentação como forma de definição de blocos de código. A comunidade Python preza muito pela legibilidade do código e possui dois elementos que reforçam ainda mais essa questão: PEP-8 e *The Zen of Python*.

O PEP-8 [9] é um guia de estilos de código Python que é amplamente empregado e existem diversas ferramentas para checá-lo automaticamente. O *The Zen of Python* [8] é um pequeno texto que fala muito sobre o estilo de programação em Python.

Apesar de ser uma linguagem interpretada, existe um processo de compilação transparente que transforma o código texto em bytecode, que, por sua vez, é interpretado por uma *virtual machine* (VM). A implementação padrão da linguagem Python é chamada de CPython e, apesar de existirem outras implementações da especificação, é nesta que vamos focar, porque é a que hoje implementa a versão 3.x, que será a base deste livro.

The Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one – and preferably only one – obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than ***right*** *now.*
If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Tradução

Bonito é melhor que feio.
Explícito é melhor que implícito.
Simples é melhor que complexo.
Complexo é melhor que complicado.
Linear é melhor que aninhado.
Esparso é melhor que denso.
Legibilidade conta.
Casos especiais não são especiais o suficiente para quebrar regras.
Embora praticidade prevaleça sobre pureza.
Erros nunca devem ser silenciados.
A não ser explicitamente.
Diante de uma ambiguidade, não caia na armadilha do chute.
Deve existir um – e preferencialmente um – jeito óbvio de se fazer algo.
Embora possa não parecer óbvio a não ser que você seja holandês.
Agora é melhor que nunca.
Embora nunca normalmente seja melhor que exatamente **agora**.
Se a implementação é difícil de explicar, ela é uma má ideia.
Se a implementação é fácil de explicar, talvez seja uma boa ideia.
Namespaces são uma grande ideia – vamos usá-los mais!

PEP-8

É importante ler a PEP pelo menos uma vez, pois há vários detalhes interessantes. Certos pontos são considerados indispensáveis por quase toda comunidade. Eis alguns tópicos importantes que vale destacar:

- Use 4 espaços para indentação;
- Nunca misture `Tabs` e espaços;

- Tamanho máximo de linha é 79 caracteres;
- `lower_case_with_underscore` para nomes de variáveis;
- `CamelCase` para classes.

O guia tem mais itens sobre indentação, espaçamento, sugestões de como usar *imports* e muito mais. A grande maioria dos programadores Python acaba adotando grande parte da PEP-8 [9] no seu dia a dia. Por isso, a sua leitura é fortemente recomendada. Você encontra a PEP-8 disponível em http://www.python.org/dev/peps/pep-0008/.

1.2 De Python 2 para Python 3

Durante o livro, abordaremos a versão 3 do Python. No entanto, é preciso saber que existem algumas diferenças importantes entre ela e a versão 2. Uma delas é a definição de *strings* e de *unicodes*. Em Python 2, strings e bytes são um (e o mesmo) tipo, enquanto unicode é um outro. Já no Python 3, strings são unicodes e os bytes são outro tipo.

Se você não faz ideia do que sejam unicodes, aqui vai uma breve explicação: em idiomas como o português, temos letras que podem ser acompanhadas de acentos (por exemplo, `é`). Tradicionalmente, esses caracteres não têm representação direta na tabela ASCII. Quando um caractere está nessa tabela, ele normalmente pode ser representado com apenas 1 byte de memória. Em outros casos, como em outros idiomas, alguns caracteres precisam de mais de 1 byte para serem representados. Criou-se, então, um padrão chamado *unicode* [2], no qual foram definidos *code points* – em uma simplificação, podemos entender como números – para diversos caracteres de várias línguas e para alguns outros tipos especiais de caracteres.

Depois da padronização, apareceram os *codecs*, que convertem os *code points* em sequências de bytes, já que, quando se trata de unicodes, não temos mais a relação de 1 caractere para 1 byte. O codec mais famoso é o UTF-8 [14], criado com objetivo de ter a melhor compatibilidade possível com o padrão ASCII.

No Python 2, como já dito, o tipo string é o mesmo que o tipo byte e um objeto unicode deve ser convertido para string usando um codec como UTF-

8. No Python 3, ou seja, no nosso caso, as strings são unicodes por padrão. Quando é necessário obter o conjunto de bytes que a formam, é preciso converter com um codec. Isso será visto com um pouco mais de detalhes mais à frente.

Em Python 3 existe apenas um tipo `int` que se comporta basicamente como o tipo `long` do Python 2. Tal mudança geralmente não é percebida na migração de uma versão para outra.

Além disso, usaremos um módulo novo com funções de estatística, chamado `statistics`. Outra novidade do Python 3 é o módulo `asyncio`, que trata de I/O assíncrono. Entretanto, sua aplicação exige mais conhecimento prévio do leitor a respeito de I/O, loop de eventos, programação assíncrona e outros assuntos mais avançados que estão fora do escopo deste livro. Mas, mesmo assim, esse módulo merece menção pela sua importância.

No geral, muitos consideram que a linguagem ficou mais consistente. Outras diferenças serão exemplificadas com códigos ao longo do livro. Na prática, para quem está iniciando com Python, elas não serão tão importantes e só serão destacadas quando pertinente.

1.3 Diversos interpretadores e mesma linguagem

Python possui diversas implementações. Aqui vamos focar na implementação que podemos chamar de referência, *CPython*. Essa implementação é feita em C e é, de longe, a mais amplamente utilizada. Outra implementação muito famosa e poderosa é chamada *PyPy*, um interpretador Python escrito em Python, que conta com um poderoso *JIT Compiler* (*Just-in-time compiler*), que melhora o desempenho de muitos programas e já é usado em produção por algumas empresas. É um projeto bem complexo e ousado, mas que vem mostrando bons resultados e um crescimento em sua adoção, assim como uma participação maior por parte da comunidade.

Ainda temos *Jython*, a implementação de Python em Java, que é mais usada por projetos em Java que querem oferecer uma linguagem mais simples para algumas tarefas, em um contexto de execução de uma máquina virtual Java. Seguindo um modelo semelhante, temos o *IronPython*, que é a imple-

mentação de Python para a plataforma .NET.

1.4 Preparando o ambiente

O início de tudo é o site http://python.org. Os principais links relacionados à programação, documentação e downloads estão lá. A documentação é boa e bem-organizada, e cobre diversos aspectos como instalação, biblioteca padrão, distribuição etc.

Se você usa Windows, consegue fazer o download do arquivo `.msi` e instalar facilmente. Já a instalação padrão da maioria das distribuições OSX/Linux instala Python como padrão, entretanto nem sempre será a versão 3. Caso não seja, existem dois caminhos: instalar com o mecanismo de pacotes do sistema (por exemplo, `apt-get` no Ubuntu, ou `brew` no OSX); ou compilar do código-fonte e instalar.

Para maiores detalhes, existe bastante material na internet com explicações mais detalhadas para outras plataformas e também como compilar a partir do código-fonte.

1.5 Primeiro passo: abra o interpretador

O primeiro passo é invocar o interpretador, executando `python`, geralmente usando um console. O que abre é um console Python interativo, no qual podemos escrever códigos e ver os resultados imediatamente. Existe um cabeçalho que diz qual é a versão e, logo depois, o ponteiro do console interativo.

O console interativo facilita muito o processo de aprendizagem, porque permite que trechos de código sejam executados em um ciclo de avaliação de expressões contínuo.

```
$ python
Python 3.4.1 (default, Aug 24 2014, 21:32:40)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

INVOCANDO O INTERPRETADOR

Em algumas instalações, o interpretador pode estar no arquivo `python3.3` em vez de `python`.

1.6 Primeiras explorações

Dentro do console interativo, as expressões já podem ser avaliadas, funções invocadas e módulos importados, exatamente como em um programa normal.

```
>>> 1
1
>>> 3 / 2
1.5
>>> print("Hello World")
"Hello World"
```

Após confirmar o comando com o `Enter`, o resultado é exibido logo abaixo.

Note que o console serve mais para exploração do que para elaboração de programas. Para executar programas, basta utilizarmos `python nome_do_programa.py`. A extensão `.py` é normalmente usada para os programas em Python. Inicialmente, vamos considerar que nossos programas sempre estão no mesmo arquivo até falarmos sobre módulos.

1.7 Próximos passos

Como em todo aprendizado de uma linguagem de programação, o primeiro passo é instalar o compilador ou interpretador. Depois de instalar o Python 3.3, 3.4 ou superior, já podemos abrir o console interativo e iniciar as primeiras explorações.

Esse processo de exploração do console interativo acaba tornando-se parte do dia a dia do programador Python. O console é uma forma muito rápida de testar trechos de código e consultar documentação. Além do

console padrão, existem muitos outros com mais recursos, como *Ipython* (http://ipython.org/) e *bpython* (http://bpython-interpreter.org/) . O projeto Ipython é amplamente utilizado e sua instalação é sugerida. Ele possui diversos recursos, incluindo *auto complete* de objetos no próprio console, entre outras coisas.

Esse console interativo também possui uma ferramenta de ajuda/documentação que pode ser usada chamando `help()` dentro do console. No próximo capítulo, vamos começar o estudo da manipulação dos tipos básicos de Python. Dessa forma, daremos um passo importante para o aprendizado da sua linguagem e de como usá-la.

Embora seja comum abrir o modo interativo múltiplas vezes durante o desenvolvimento, software em Python não é feito nesse console. Mais à frente, veremos como criar pacotes e distribuir software escrito em Python, e usaremos o modo interativo para estudo e prototipação.

Nas próximas etapas, vamos falar um pouco mais de características gerais da linguagem e iniciar a parte prática.

Capítulo 2

Aprendendo Python na prática: números e strings

Na prática, todos os programas vão manipular textos ou números. Por isso, precisamos saber trabalhar com esses tipos fundamentais. No universo Python, os termos que usaremos são *números* e *strings*. Manipulá-los é relativamente simples até para não programadores. Primeiro, vamos conhecer os conceitos básicos, para depois explorar alguns exemplos e explicar mais detalhes dos seus comportamentos.

2.1 Números

Em programas de computador, quase sempre vamos manipular números. Um dos primeiros passos para aprender uma linguagem de programação é sa-

ber como operar seus números. Cada linguagem disponibiliza, por padrão, alguns tipos de números embutidos que estão presentes em sua sintaxe; há outros em módulos podem ser importados e são manipulados por meio de objetos e APIs.

Python 3 possui três tipos de números embutidos: `int`, `float` e `complex`.

Inteiros são virtualmente ilimitados [3] e podem crescer até o limite da memória. Outro tipo muito conhecido de números é o ponto flutuante, chamado de *float*. Em Python, no interpretador padrão CPython, os `floats` são implementados usando o tipo `double` do C [13]. Entretanto, em outros interpretadores – como Jython –, isso é feito por meio do tipo flutuante de precisão dupla da plataforma abaixo, no caso a JVM.

Os números complexos também estão embutidos na linguagem, mas sua aplicação não é tão comum no dia a dia. Eles têm a parte imaginária e a real, sendo que cada uma delas é um `float`. Os números complexos, por exemplo, dão resposta como a raiz quadrada de um número negativo e são usados em domínios como engenharia elétrica, estatística etc.

Números inteiros são escritos normalmente como em sua forma literal (por exemplo, `100`) e também podem ser gerados usando a função `int()` como em `int('1')`, no qual geramos o número inteiro `1` a partir de uma string. Números de ponto flutuante têm o caractere `.` – como em `2.5` –, para separar as casas decimais. Também podem ser gerados pela função `float()` – como em `float(1)` ou `float('2.5')` –, e são aceitas na função `float()` as strings `nan` e `inf` com prefixos opcionais `+` e `-` – por exemplo, `float('+nan')`. Os números complexos têm o sufixo `j` ou `J`, como em `1+2J`. Para acessar a **parte real** e a **parte imaginária**, use `1+2j.real` ou `1+2j.imag`.

Formas literais de escrever os números básicos em Python:

```
>>> 1     #int
1
>>> 1.0   #float
1.0
>>> 1.    #também float
1.0
```

```
>>> 1+2j #complex
1+2j
```

Gerando os números por meio das funções embutidas:

```
>>> int(1.0)
1
>>> int('9')
9
>>> float(1)
1.0
>>> float('9.2')
9.2
>>> float('-inf')
-inf
>>> float('+inf')
inf
>>> float('nan')
nan
>>> complex(1, 2)
(1+2j)
```

Ints e longs unificados no Python 3

Diferente do Python 2, no qual existem 2 tipos de inteiros, `int` e `long`, o Python 3 contém **apenas** o `int`, que se comporta como o `long` do Python 2. Anteriormente, um inteiro poderia tornar-se um longo se ultrapassasse o limite de `sys.maxint`. A PEP-237 [7] foi responsável por essa unificação.

A seguir, alguns exemplos de números e manipulações simples que você pode fazer para testar seu ambiente:

```
>>> 3 + 2
5
>>> 3 + 4.2
7.2
```

```
>>> 4 / 2
2.0
>>> 5 / 2
2.5
>>> 5 // 2
2
>>> complex(1, 2) + 2
(3+2j)
>>> complex(2, 0) + 0+1j
(2+1j)
>>> 2 + 0+1j
(2+1j)
```

Alguns operadores podem ser novos, como o `//`. Vamos estudá-los em seguida!

Operadores aritméticos

O conjunto base de operadores aritméticos com números em Python é: `x + y` (adição), `x - y` (subtração), `x / y` (divisão em ponto flutuante), `x // y` (divisão descartando parte fracionária), `x * y`, (multiplicação), `x % y` (resto), `-x` (negação) e `x ** y` (potência).

```
>>> 1 + 2
3
>>> 3 - 1
2
>>> 10 / 2
5.0
>>> 10 // 3
3
>>> 10 * 2 + 1
21
>>> 10 % 3
1
>>> -3
-3
>>> 2 ** 8
256
```

Operadores de bits

O conjunto base de operadores com bits é: `x | y` (ou), `x ^y` (ou exclusivo), `x & y` (e), `x << y` (x com y bits deslocados à esquerda), `x >> y` (x com y bits deslocados à direita) e `~x` (inverso em bits).

```
>>> 1 | 0
1
>>> 1 | 5
5
>>> 1 ^ 5
4
>>> 4 & 1
0
>>> 1 << 2
4
>>> 4 >> 2
1
>>> ~4
-5
```

Operações misturando tipos diferentes e as regras de coerção

Como Python é uma linguagem dinâmica, muitos programas podem operar números de tipos diferentes em uma mesma operação. Podemos definir o valor de um imposto com um `float` e aplicar a um valor inteiro. Veja o exemplo:

```
>>> 100 * 1.3 # preço mais 30%
130.0
```

Para isso, existe uma política de coerção de números que define qual o tipo resultante de uma operação que mistura tipos diferentes de números.

Se operarmos um `int` e um `float`, vamos ter como resultado um `float`. Se operarmos um `int` ou `float` com um `complex`, vamos ter como resultado um `complex`.

Explorando as operações por conta própria:

```
>>> type(1 + 2.0)
<class 'float'>
```

```
>>> type(1 + 2J)
<class 'complex'>
>>> type(1.0 + 2J)
<class 'complex'>
>>> type(1.0 + 1.0)
<class 'float'>
```

Função type()

A função `type(obj)`, com apenas um parâmetro, retorna o tipo (ou classe) de um objeto. Na dúvida, use-a e descubra o tipo de um objeto referenciado por uma variável. Para realizar testes em condicionais, é recomendado o uso da função `isinstace(obj, class)`. Para meras verificações em explorações nos terminais, use `type()` quando tiver dúvida sobre o tipo de uma variável.

2.2 Como manipular texto em Python

A string é um tipo (ou classe) fundamental em muitas linguagens. Nas de alto nível, sua manipulação geralmente é fácil. Python possui muitas conveniências na manipulação de strings que vamos entender melhor daqui para a frente. Além de manipular, vamos entender como Python 3 representa internamente e quais as implicações práticas da sua implementação atual de strings.

Diferente de linguagens como C, não existe o tipo *char*, que é um tipo inteiro, de 1 byte, usado pra representar valores da tabela ASCII. Em Python, existem apenas strings, mesmo que contenham apenas um caractere ou sejam vazias. Portanto, podemos interpretar que elas são sequências de caracteres de tamanho 0 até o máximo suportado.

2.3 Criando e manipulando texto: strings

Python tem formas muito convenientes de declarar strings e de formatá-las. Vamos ver alguns exemplos:

```
# coding: utf-8

"Copa 2014"

'Copa do mundo 2014'

'''2014 - Copa do mundo
'''

"copa 'padrão fifa'"

'copa "padrão fifa"'
```

Os formatos principais são com ' e ", sendo que ao usar um, o outro pode ser usado internamente como nos dois últimos exemplos. Outra opção são as *multiline strings*, com três aspas simples ou três aspas duplas. Ela é muito empregada em formatações de saídas de console. Veja no código:

```
# coding: utf-8

print("""\
Uso: consulta_base [OPCOES]
     -h                         Exibe saída de ajuda
     -U url                     Url do dataset
""")
```

O que gerará a seguinte saída:

```
Uso: consulta_base [OPCOES]
     -h                         Exibe saída de ajuda
     -U url                     Url do dataset
```

Repare na \ no início da string que escapa o \n da quebra de linha.

As strings literais separadas apenas por espaço serão implicitamente convertidas em uma única string literal.

```
("Copa" "2014") == "Copa2014"
```

É possível quebrar strings longas que não são *multiline* apenas ao quebrar a linha, de preferência indentando-as.

```
input('Em qual cidade o legado da Copa foi mais relevante '
      'para a populacao?')
```

Prefixo de literal unicode

Ao ler código Python, certamente você ainda encontrará muitas strings com o prefixo `u`, por exemplo `u"minha string"`. Essa sintaxe surgiu na versão 2 para dizer que uma string está escrita em unicode. Já na versão 3.3, a sintaxe `u"minha string"` passou novamente a ser aceita, para facilitar a portabilidade de programas escritos na versão 2, nos quais os literais unicode devem ter obrigatoriamente o prefixo, como em `u'string'`. Anteriormente, nas versões 3.0 até 3.2, as strings com prefixo `u` eram um erro de sintaxe.

String é uma sequência

Em strings podemos acessar os elementos *code points* usando um índice e a notação `variavel[ index ]`. O índice varia de 0 até o tamanho da string menos 1. Se ele for negativo, a contagem é na ordem inversa. O quadro a seguir ilustra melhor:

```
.
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Em Python, o termo *sequência* tem um significado especial. Para que um objeto seja uma *sequência*, como uma string, ele deve atender alguns requisitos. A seguir, veremos a aplicação de alguns conceitos de *sequência* em strings Python. Três das suas manipulações fundamentais são: saber tamanho, acessar um item por posição e acessar trechos por posições.

Para saber o tamanho, usamos `len(string)`. Para acesso por índice, utilizamos `variavel[indice]`, ou para acesso de trechos, a *slice notation*, como em `minha_str[1:2]` na prática:

```
>>> st = "maracana"
>>> st[0]
'm'
>>> st[1:4]
'ara'
>>> st[2:]
'racana'
>>> st[:3]
'mar'
>>> len(st)
8
```

Por ser uma sequência, além do acesso por índices e *slices*, podemos executar outras operações como: `x in y`, se x está em y; `x not in y`, se x não está em y; `x + t`, concatenação de x com y; e `x * y`, y repetições de x.

```
>>> "m" in "maracana"
True
>>> "x" not in "maracana"
True
>>> "m" + "aracana"
'maracana'
>>> "a" * 3
'aaa'
```

Imutabilidade: novas strings criadas a partir de outras strings

As strings são sequências imutáveis de code points [10]. Isso significa que elas têm seu valor definido na criação e que as novas sempre são criadas a partir das operações com elas. Se tentarmos mudar o valor de uma posição ou pedaço de uma string, vamos receber um erro:

```
>>> minha_str = "livro python 3"
>>> minha_str[13] = "2"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Algumas formas possíveis:

```
>>> minha_str = "livro python 3"
>>> minha_str = minha_str[0:13] + "2"
>>> minha_str
'livro python 2'
>>> minha_str = "livro python 3"
>>> minha_str = minha_str.replace("3", "2")
>>> minha_str
'livro python 2'
```

Principais métodos

Temos a documentação de todos os métodos de strings da biblioteca padrão disponível em http://docs.python.org/3.3/library/stdtypes.html#string-methods. Alguns muito comuns são: `capitalize`, `count`, `endswith`, `join`, `split`, `startswith` e `replace`.

```
>>> "maracana".capitalize()
'Maracana'
>>> "maracana".count("a")
4
>>> "maracana".startswith("m")
True
>>> "maracana".endswith("z")
False
>>> "copa de 2014".split(" ")
['copa', 'de', '2014']
>>> " ".join(["Copa", "de", "2014"])
'Copa de 2014'
>>> "copa de 2014".replace("2014", "2018")
'copa de 2018'
```

Interpolação de string

Em Python, podemos interpolar strings usando `%` ou a função `.format()`.

> Python também implementa o formato `printf` de impressão, documentado em http://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting, assim como um formato próprio, documentado em http://docs.python.org/3/library/string.html#formatstrings.

```
>>> "%d dias para copa" % (100)
'100 dias para copa'
>>> "{} dias para copa".format(100)
'100 dias para copa'
>>> "{dias} dias para copa".format(dias=100)
'100 dias para copa'
```

Podemos configurar espaçamentos e alinhamentos, da seguinte forma:

```
>>> '{:<60}'.format('alinhado à esquerda, ocupando 60 posições')
'alinhado à esquerda, ocupando 60 posições                   '
>>> '{:>60}'.format('alinhado à direita, ocupando 60 posições')
'                    alinhado à direita, ocupando 60 posições'
>>> '{:^60}'.format('centralizado, ocupando 60 posições')
'             centralizado, ocupando 60 posições             '
>>> '{:.^60}'.format('centralizando ao alterar caractere
de espaçamento')
'......centralizando ao alterar caracter de espaçamento.......'
```

2.4 Como Python representa strings internamente?

No Python 3, todas as strings são representações em unicode code points [10]. Elas podem ter caracteres com acentos (por exemplo, `"é"`) ou qualquer outro unicode válido, como caracteres do japonês.

A questão é que muitos caracteres, inclusive da língua portuguesa, não podem ser representados em apenas 1 byte. Então, foi necessária uma diferenciação entre os objetos que utilizam um byte por letra, que seriam apenas as letras e caracteres da tabela ASCII, e os objetos que usam caracteres que necessitam de mais de 1 byte pare serem representados.

No Python 2, uma string é uma sequência de bytes, logo não podem representar unicodes a não ser que estes sejam convertidos para bytes com um codec. Acontece que, nessa versão, muitas vezes uma conversão implícita era feita usando UTF-8, e strings e bytes são o mesmo objeto, enquanto unicode é outro. E no Python 3, strings e unicodes são a mesma coisa (`string`) e os bytes um outro objeto.

Assim, no Python 2, a string `"é"` é implicitamente convertida para sua representação em bytes, usando UTF-8. Nesse caso, o seu tamanho seria `len("é") == 2`. Porém, no Python 3.3, a mesma string teria tamanho 1, porque ela contém apenas um code point e representa isso semanticamente. Logo, o tamanho seria `len("é") == 1`.

Isso poderia representar uma perda de espaço se os caracteres ASCII – os que conseguem ser representados em 1 byte – tivessem que ocupar 2 bytes em memória. Felizmente, a PEP-393 (*Flexible String Representation* [6]) faz com que Python 3 represente os code points de maneira eficiente em memória.

Em algumas situações específicas pode ser necessário converter explicitamente strings [10] para bytes. Isso deve-se ao fato de que um unicode pode ser convertido para bytes por meio de diversos codecs, sendo os mais famosos o UTF-8 e o UTF-16. O mesmo unicode torna-se uma sequência de bytes diferente, dependendo do codec usado na conversão.

Existe uma grande discussão sobre essa mudança da representação em code points das strings, mas não vamos entrar em detalhes neste livro. Na visão do autor, a comunidade ainda está digerindo os efeitos dessa mudança. Entre 2013 e 2014, muita coisa foi portada para Python 3.3, mas a adoção ainda não é em larga escala, embora seja bem relevante.

2.5 Conclusão

Neste capítulo, vimos os aspectos fundamentais de números e strings (texto) em Python e algumas de suas características. Aprendemos a escrevê-los em sua forma literal ou usando funções auxiliares para gerá-los a partir de outros dados, como gerar inteiros ou flutuantes a partir de strings.

Vimos também um pouco sobre detalhes da implementação de strings em Python e as diferenças que esse objeto sofreu na migração da família 2

para a 3, e aprendemos alguns exemplos de métodos de strings disponíveis na biblioteca padrão.

Ao longo do livro, exploraremos a manipulação de números e strings no contexto de um aplicativo que vamos desenvolver. No próximo capítulo, vamos aprender mais ferramentas que nos ajudarão a estruturar nossos programas, como: condicionais, coleções e loops.

Capítulo 3

Manipulações básicas

As manipulações básicas são as manipulações aritméticas simples e as operações com objetos e funções, como declarações e chamadas. Para passar os conceitos iniciais de manipulações aritméticas, vamos ver uma pequena calculadora a seguir.

3.1 Uma calculadora: o exemplo revisado mais comum

Nosso primeiro programa vai aplicar uma taxa `float` sobre um valor qualquer `int`. Se essa mesma taxa for utilizada em diversas partes de um programa, faz sentido guardar o valor em uma variável. Se a taxa mudar, basta trocar o valor declarado da variável e o resto do programa permanece igual. Repare que ainda não vimos como criar variáveis. Conceitualmente, essa ta-

refa chama-se **atribuição**. Em Python, uma atribuição se faz `variavel = expressão`. Vamos ver exemplos reais:

```
>>> imposto = 0.27
>>> salario = 5000
>>> print(salario - (salario * imposto))
3650.0
```

Nesse exemplo, `salario` é um `int`, e `imposto` é um `float`. Repare que o resultado é um `float`, como vimos nas regras de coerção.

Podemos aplicar formatação de strings e expressões para termos um resultado mais atraente:

```
>>> imposto = 0.27
>>> salario = 3000
>>> print("Valor real: {0}".format(salario -
                                    (salario * imposto)))
Valor real: 2190.0
```

3.2 Pegando dados no terminal

Em Python 3, usamos a função `input([prompt])` para capturar um *input* do usuário. Com esse recurso, podemos perguntar ao usuário qual o valor do imposto cujo salário real ele deseja calcular.

Input no Python 2 e 3

Na versão 2, a função para capturar a entrada do usuário é `raw_input([prompt])`, enquanto `input([prompt])` avalia expressões.

No nosso primeiro programa, vamos usar as funções: `print()`, para imprimir na tela o feedback; `input()`, para pegar a entrada do usuário; e `int()` e `float()`, para converter o input de `string` para `int` e `float`, respectivamente.

Coloque o código a seguir em um arquivo e salve-o como `salario_real.py`. Em seguida, execute-o no console `python salario_real.py`.

```
salario = int(input('Salario? '))
imposto = float(input('Imposto em % (ex: 27.5)? '))
print("Valor real: {0}".format(salario - (salario *
                                          (imposto * 0.01))))
```

Esse pequeno programa mostra muito sobre Python: não tem tipos explícitos, não precisa obrigatoriamente de uma função *main* – embora seja uma boa prática ter uma – e manipula tipos diferentes em uma operação aritmética. Baseando-se nesse exemplo, vamos evoluir nosso pequeno programa introduzindo condicionais e loops.

3.3 Comparações: maior, menor, igual e outras

Em Python, existem 8 operadores de comparação. Neste início do livro, seis deles serão muito importantes, tendo eles uma semântica simples e muito semelhante à de outras linguagens.

Os operadores são:

- < – menor que;
- <= – menor ou igual que;
- > – maior que;
- >= – maior ou igual;
- == – igual;
- != – não igual.

O comportamento dos operadores é bem intuitivo e segue essa lógica apresentada. Veja alguns exemplos:

```
>>> 1 >= 1
True
>>> 2 < 1
False
>>> 9 == 9
True
>>> 9 != 8
True
>>> 2 <= 3
True
```

Objetos de tipos distintos nunca são considerados iguais, exceto números. Os números, por sua vez, podem ser comparados entre si, exceto pelo número complexo, que, quando comparado a outro tipo de número, gera um erro do tipo `TypeError`. Novamente, vamos ver exemplos:

```
>>> 1 == 1.0
True
>>> 10 > 1j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() > complex()
```

Instâncias de classes podem ser comparadas, mas veremos isso mais à frente. Por enquanto, queremos fazer comparações com tipos básicos para suprir as nossas necessidades iniciais.

3.4 Condicionais: if, elif & else

Se considerarmos que grande parte das pessoas no Brasil paga a mesma cota de impostos, podemos dizer que existe um valor padrão e usá-lo se o usuário não souber informar quanto paga. Para isso, precisamos aplicar um `if` e saber se ele deixou o input vazio. Caso tenha deixado, utilizaremos o valor padrão da taxa. Caso o input não esteja vazio, vamos usá-lo como valor da taxa de imposto digitada pelo usuário.

```
salario = int(input('Salario? '))
imposto = input('Imposto em % (ex: 27.5)? ')
```

```
if imposto == '':
    imposto = 27.5
else:
    imposto = float(imposto)

print("Valor real: {0}".format(salario - (salario *
                                        (imposto * 0.01))))
```

A função input() retorna uma string vazia se o usuário teclar Enter sem nenhuma entrada. Portanto, imposto terá um valor igual a "" (string vazia) e o primeiro bloco será executado, enquanto o bloco do else não será.

Comando if

Na teoria, um if é um comando que avalia uma **expressão** e escolhe um bloco para ser executado, de acordo com o resultado dessa avaliação. A expressão, nesse caso, é imposto == '', tendo uma variável sendo comparada com o operador de igualdade == com um valor literal de uma string vazia. Semanticamente, uma string vazia em um bloco if equivale a False, assim como uma lista vazia. Poderíamos mudar o teste do if anterior para algo como o código a seguir:

```
salario = int(input('Salario? '))
imposto = input('Imposto em % (ex: 27.5)? ')
if not imposto:
    imposto = 27.5
else:
    imposto = float(imposto)

print("Valor real: {0}".format(salario - (salario *
                                        (imposto * 0.01))))
```

As expressões podem conter operadores explicitamente booleanos ou não. Como falamos, em um comando if uma expressão de lista vazia é considerado como False. No exemplo anterior, a expressão not imposto na verdade é o mesmo que not "" e que é interpretada como um booleano, no caso, True, já que no comando if uma string vazia é considerado como False.

Indentação dos blocos de código

Aqui uma outra característica marcante de Python fica muito clara: a definição dos limites do início e fim dos blocos `if`, `elif` e `else` são feitas com indentação. Sempre que andamos 4 espaços para a direita, seguindo a PEP-8 [9], estamos definindo um novo bloco. Já quando voltamos os 4 espaços, significa que aquele bloco terminou.

Assim como em outras linguagens, também temos definidos os outros elementos como: `elif` e `else`. O `elif` avalia uma outra expressão e é executado caso esta seja avaliada como verdadeira. No caso de nenhuma expressão de `if` ou `elif` ser verdadeira, o bloco do `else` é executado, se existir. Repare que o corpo dos blocos das condicionais encontra-se 4 espaços depois do canto esquerdo.

```
imposto =  float(input("Imposto: "))
if imposto < 10:
    print("Medio")
elif imposto < 27.5:
    print("Alto")
else:
    print("Muito alto")
```

Expressão if

Também existem as expressões condicionais `if`, chamados de operadores ternários. Por serem uma expressão, eles têm um valor associado e podem ser atribuídos a variáveis, diferente do comando `if`, que não tem nenhum valor associado.

Veja o exemplo:

```
>>> imposto = 0.3
>>> "Alto" if imposto > 0.27 else "Baixo"
'Alto'
>>> imposto = 0.10
>>> "Alto" if imposto > 0.27 else "Baixo"
'Baixo'
>>> valor_imposto = "Alto" if imposto > 0.27 else "Baixo"
>>> valor_imposto
'Baixo'
```

3.5 Operadores lógicos

Igual a outras linguagens, aqui também temos outras operações booleanas que podem ser usadas nas expressões avaliadas em um `if` ou até mesmo em atribuições. Elas são: `and`, `or` e `not`. Ou seja, são os operadores lógicos *e*, *ou* e negação, respectivamente, com funcionamento muito semelhante ao de outras linguagens, como C ou Java.

Ambos `and` e `or` são operadores com curto circuito. No caso do `and`, a segunda expressão só é avaliada caso a primeira seja `True` e, no caso do `or`, a segunda só é avaliada caso a primeira seja `False`.

O exemplo anterior poderia ser melhorado com esses operadores:

```
imposto =  float(input("Imposto: "))
if imposto < 10.:
    print("Baixo")
elif imposto >= 10. and imposto <= 27.:
    print("Médio")
elif imposto > 27. and imposto < 100:
    print("Alto")
else:
    print("Imposto inválido")
```

3.6 Loops com while

Em muitas linguagens, quando vamos implementar loops, usamos o comando `while`. O `while` em Python também avalia uma expressão e executa um bloco até que esta seja avaliada como falsa, tenha uma chamada `break` ou levante uma exceção sem tratamento.

Vamos permitir que nosso programa calcule para um mesmo salário diversos valores com imposto descontado, utilizando o comando `while`. No primeiro exemplo, vamos sair com a avaliação da expressão retornando falso.

```
salario = int(input('Salario? '))
imposto = 27.
while imposto > 0.:
    imposto = input('Imposto ou (0) para sair: ')
    if not imposto:
```

```
        imposto = 27.
    else:
        imposto = float(imposto)
    print("Valor real: {0}".format(salario - (salario *
                                              (imposto * 0.01))))
```

O loop pode ser interrompido sem execução de parte do bloco com um comando `break`, como podemos ver no exemplo a seguir:

```
salario = int(input('Salario? '))
imposto = 27.
while imposto > 0:
    imposto = input('Imposto ou (s) para sair: ')
    if not imposto:
        imposto = 27.
    elif imposto == 's':
        break
    else:
        imposto = float(imposto)
    print("Valor real: {0}".format(salario - (salario *
                                                imposto * 0.01)))
```

O `while` é muito usado quando não se sabe previamente quando o loop deve terminar. Porém, em muitos casos, queremos percorrer coleções de elementos. Obviamente, podemos realizar essa operação com o comando `while`, mas em Python existe uma forma melhor de se fazer isso.

Primeiro, vamos ter um pequeno contato com essas coleções e, depois, veremos como realizamos esse tipo de operação de percorrer todos elementos de uma coleção.

3.7 Primeira estrutura de dados: lista

Em Python, a sintaxe da lista é muito enxuta: `[]`. Os itens são separados por `,` (vírgula), como em `[1, 2, 3]`. Listas não precisam ter elementos do mesmo tipo, como em linguagens estaticamente tipadas, e podem misturar livremente tipos diferentes, embora não seja algo comum. Alguns exemplos de declarações de listas:

```
>>> [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
>>> ["salario", "imposto"]
['salario', 'imposto']
>>> [1, "salario"]
[1, 'salario']
>>> [[1, 2, 3], "salario", 10]
[[1, 2, 3], 'salario', 10]
```

Lista também é uma *sequência* em Python, ou seja, podemos perguntar seu tamanho e acessar elementos por índice ou trechos (*slices*).

Para saber o tamanho, usamos `len(lista)`. Já para acessar por índice, usamos `lista[indice]` e por trechos, utilizamos a *slice notation*, como em `lista[1:2]`. Exemplos:

```
>>> lista = ['impostos', 'salarios', 'altos', 'baixos']
>>> lista[0]
'impostos'
>>> lista[-1]
'baixos'
>>> lista[2:4]
['altos', 'baixos']
```

Listas são mutáveis, logo, podemos realizar atribuições em índices, ou em trechos:

```
>>> lista = ['impostos', 'salarios', 'altos', 'baixos']
>>> lista[2] = "Altos"
>>> lista[3] = "Baixos"
>>> lista
['impostos', 'salarios', 'Altos', 'Baixos']
>>> lista[0:2] = ["Impostos", "Salarios"]
>>> lista
['Impostos', 'Salarios', 'Altos', 'Baixos']
```

Ifs e listas

Em Python, o `if` avalia listas vazias como falso, e isso é muito empregado na prática. Portanto, acostume-se. Veja um exemplo a seguir:

```
lista = []
if lista:
    print("Nunca sou executado")
else:
    print("Sempre sou executado")
```

3.8 Loop pythônico com for e listas

Percorrer elementos de uma coleção é algo que todo programador precisa. Porém, nem todas as linguagens possuem uma forma simplificada para realizar essa tarefa. Na programação imperativa com Python, o comando `for` é capaz de tornar essa tarefa trivial. Dado o nome de uma variável e uma lista, o `for` é realizado da seguinte forma:

```
>>> impostos = ['MEI', 'Simples']
>>> for imposto in impostos:
...     print(imposto)
...
MEI
Simples
```

Comando for em detalhe

O `for` faz, para cada elemento da lista, uma atribuição do elemento corrente à variável definida no comando, e executa o bloco de código associado a essa variável disponível. Assim como o `while`, ele também pode ser parado por um `break` ou por uma exceção não tratada. Um outro recurso que também é compatível com o `while` é a palavra reservada `continue`. Esse comando faz com que a execução do bloco vá direto para a próxima iteração. Veja o exemplo para ficar mais claro:

```
>>> impostos = ['MEI', 'Simples']
>>> for imposto in impostos:
...     if imposto.startswith("S"):
...         continue
...     print(imposto)
...
MEI
```

Para avançar com o `for`, vamos analisar o próximo problema. Se quisermos imprimir os números de 0 até 10, poderíamos ter:

```
>>> lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for i in lista:
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
```

Mas e se quiséssemos imprimir os números de 0 até 100? Certamente não temos que criar manualmente uma lista com esses elementos. Vamos ver mais elementos que combinam com o `for`.

3.9 Percorrendo intervalos de zero até N com range()

Até o momento, não vimos como percorrer intervalos de zero até N como no clássico `for(i = 0; i < n; i++)`, que existe em C, Java, JavaScript e em muitas outras linguagens. Vamos para o exemplo:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Veja como é curioso o resultado da avaliação de `range(5)`:

```
>>> range(5)
range(0, 5)
```

Em Python 3, um objeto retornado por `range()` é compatível com o `for`, porém não é uma lista. Esse objeto do tipo `range` é um que chamamos de iterável, definido pela PEP-234 [5]. Ou seja, conseguimos dele um iterador que a cada chamada retorna um valor diferente, até que uma exceção `StopIteration` seja levantada e o `for` terminado. Mais à frente, aprenderemos a criar nossos objetos iteráveis que poderão ser utilizados junto com o `for`.

Range no Python 2

No Python 2, a função `range()` retorna uma lista, o que também é muito intuitivo, mas implica em uma lista de tamanho N ser construída para ser retornada. Se o valor N for muito grande (por exemplo, 1.000.000), podem ser ocupados, aproximadamente, 31.074 MB de memória.

Já no Python 3, a função `range` retorna um objeto iterável que, por sua vez, retorna cada um dos elementos que fariam parte da lista, um de cada vez. Dessa forma, não precisaríamos de uma lista grande em memória para executar um `for` sobre `range(1000000)`.

3.10 Enumerando coleções com for e função enumerate

Muitas vezes, queremos enumerar elementos de uma coleção. Ou seja, além do elemento, queremos o seu índice. Com o comando `for` e a função `enumerate`, isso torna-se trivial:

```
>>> impostos = ['MEI', 'Simples']
>>> for i, imposto in enumerate(impostos):
...     print(i, imposto)
...
0 MEI
1 Simples
```

O comando `for` funciona com qualquer objeto do tipo **sequência**, como listas, strings ou com objetos iteráveis, que veremos na sequência.

3.11 Declarando funções: comando def

Em Python, funções são objetos de primeira classe (*first class objects*), portanto, podem ser passadas como parâmetros, atribuídas a variáveis, retornar outras funções e, até mesmo, terem atributos próprios. Nesta parte inicial do livro, apenas aprenderemos a declarar e chamar funções. Exploraremos as outras características mais à frente. Agora, o que vamos ver são variações na declaração e chamada de funções.

Declarações de função são feitas usando o comando `def`:

```
def sum(a, b):
    return a + b

c = sum(1, 3)
```

No dia a dia, duas outras características são muito comuns na manipulação de funções e estão relacionadas à declaração e chamada.

3.12 Valores padronizados de argumentos

Funções em Python podem ter valores padrão para seus argumentos. No nosso exemplo, poderíamos ter:

```
def salario_descontado_imposto(salario, imposto=27.):
    return salario - (salario * imposto * 0.01)
```

Esse valor padrão reflete diretamente na chamada:

```
>>> salario_descontado_imposto(5000)
3650.0
```

Veja que, por termos um valor padrão para um argumento, não obrigamos o usuário a informar um valor. Isso é valioso para quem escreve a função e para quem a usa. Então, sempre que houver um valor padrão, use-o.

Um detalhe muito importante é que o valor padrão do argumento é avaliado na **hora da avaliação** da declaração da função, e *não* na **hora da chamada**. Não se esqueça dessa característica!

3.13 Parâmetros nomeados

Em Python, outra característica da chamada de função são os parâmetros nomeados. A princípio, eles são usados para mudar os valores dos argumentos com valor padrão.

```
>>> salario_descontado_imposto(5000, imposto=0.10)
4500.0
```

3.14 Recebendo um número arbitrário de argumentos: packing & unpacking

Uma outra característica muito interessante em Python é que podemos ter funções que recebem números arbitrários de argumentos, posicionais ou nomeados. Essa característica influencia tanto na chamada da função quanto no recebimento dos parâmetros.

Como sua aplicação varia muito, vamos ver um exemplo para ilustrar melhor.

A função `date(year, month, day)` do módulo `datetime` recebe três parâmetros. Em alguma situação, poderíamos ter, em uma tupla, os valores `(2014, 10, 1)` vindos de alguma outra parte do código. Da forma como aprendemos até agora, teríamos que chamá-la da seguinte forma:

```
>>> from datetime import date
>>> d = (2013, 3, 15)
>>> date(d[0], d[1], d[2])
datetime.date(2013, 3, 15)
```

Packing

Felizmente, com o *packing*, conseguimos tornar esse caso menos verboso e mais elegante. A lógica é que se temos uma lista ou tupla com os valores que

estão na mesma ordem dos parâmetros que a função recebe, podemos usar o *packing*.

Veja o exemplo:

```
>>> from datetime import date
>>> d = (2013, 3, 15)
>>> date(*d)
datetime.date(2013, 3, 15)
```

O que aconteceu aqui é que se `date` espera os parâmetros `year`, `month`, `day` e temos uma lista ou tupla com os valores que casam essa ordem, podemos usar a sintaxe do `*tupla` para sinalizar que a coleção deve ter suas posições casadas com parâmetros recebidos.

Caso o *packing* seja sinalizado, mas a coleção tenha mais elementos que os parâmetros da função, vamos receber um `TypeError`.

O *packing* também vale para os parâmetros nomeados. Muitas vezes é comum que os parâmetros estejam já disponíveis em um dicionário. Sem o *packing*, teríamos algo como:

```
>>> def new_user(active=False, admin=False):
>>>     print(active)
>>>     print(admin)
>>>
>>> config = {"active": False,
>>>           "admin": True}
>>>
>>> new_user(config.get('active'), config.get('admin'))
False
True
```

Novamente, queremos algo mais enxuto e elegante. Podemos usar o *packing* com os parâmetros nomeados:

```
>>> def new_user(active=False, admin=False):
>>>     print(active)
>>>     print(admin)
>>>
>>> config = {"active": False,
```

```
>>>             "admin": True}
>>>
>>> new_user(**config)
False
True
```

Nesse caso, a sintaxe é `**dicionário`. O interpretador automaticamente casa os itens do dicionário com os parâmetros nomeados da função `new_user`.

Unpacking dos argumentos

O *unpacking* é o processo que é executado dentro da função, e não na chamada. Podemos usar a sintaxe `*args` ou `**kwargs` como argumentos para o *unpacking* dos parâmetros posicionais ou nomeados.

```
>>> def unpacking_experiment(*args):
...     arg1 = args[0]
...     arg2 = args[1]
...     others = args[2:]
...     print(arg1)
...     print(arg2)
...     print(others)
...
>>> unpacking_experiment(1, 2, 3, 4, 5, 6)
1
2
(3, 4, 5, 6)
```

Como a assinatura da função usa `*args`, do ponto de vista do chamador (ou *caller*), ela pode receber um número arbitrário de parâmetros, como no exemplo anterior.

O mesmo aplica-se aos parâmetros nomeados. Se usarmos `**kwargs`, o chamador pode passar quaisquer parâmetros nomeados que podem acessar `kwargs` como um dicionário e obter os valores. Veja o exemplo:

```
>>> def unpacking_experiment(**kwargs):
...     print(kwargs)
...
```

```
>>> unpacking_experiment(named="Test", other="Other")
{'other': 'Other', 'named': 'Test'}
```

Esses recursos são muito poderosos e podem ajudar em situações específicas, mas não são tão incomuns. No capítulo 8, teremos um emprego real do *packing*.

3.15 Usando código já pronto: importando módulos

Uma outra manipulação básica é a importação de módulos. Por enquanto, vamos aprender apenas como importar e usar os módulos. Mais à frente, aprenderemos como criar os nossos módulos.

A PEP-8 [9] tem recomendações sobre como estruturar o código de importação, como usar ordem alfabética, por exemplo. Neste momento do livro, o mais importante é *como importar* e *como usar* o código importado.

O comando `import` pode importar módulos ou objetos (classes e funções) para o escopo de execução do código que o executa. Quando fazemos isso, incluímos o nome usado na importação na lista de nomes disponíveis. Veja o exemplo:

```
import math
print(math.sqrt(9))
```

Nesse exemplo, importamos o módulo `math`. Logo, no contexto de execução do código, o nome `math` é a referência para o módulo. Até podemos atribuir um valor para `math`, entretanto perderíamos a referência para o módulo. Veja o exemplo:

```
import math
math = 10
print(math.sqrt(9))
Traceback (most recent call last):
  File "03_09_error.py", line 3, in <module>
    print(math.sqrt(9))
AttributeError: 'int' object has no attribute 'sqrt'
```

É possível criar um *alias* para um módulo ou um objeto importado, se quisermos usar outro nome. Veja o exemplo:

```
import math as matematica
print(matematica.sqrt(9))
```

Um outro caso de uso é importar apenas um objeto específico de um módulo. Também é uma prática muito comum. Nesse caso, utilizamos o comando `from/import`, sinalizando no `from` o módulo, e depois do `import`, informamos que objeto queremos importar. Veja o exemplo:

```
from unittest import TestCase
from unittest import mock
```

Podemos usar o recurso do *alias* com o `from/import` também:

```
>>> from math import log2 as l2
>>> print(l2(1024))
10.0
```

Agora, fechamos as manipulações básicas de importação, que já nos permitem usar módulos de terceiros ou da própria biblioteca padrão.

3.16 Conclusão

Neste capítulo, aprendemos sobre condicionais usando o comando `if`, como declarar listas de objetos Python com os colchetes `[]` e dois tipos de comandos de loop, `while` e `for`. Vimos também como podemos interromper esses loops com exceções ou usando o comando `break`, e como podemos utilizar loops com objetos iteráveis, por exemplo no caso do objeto retornado pela função `range()`.

Descobrimos como combinar `for` e `range()` para gerar um loop de inteiros em sequência, por exemplo `for(i = 0; i < n; i++)`, o básico do uso de módulos foi coberto com o comando `import` e algumas variações foram explicadas. Também aprendemos um pouco sobre o uso básico de funções e algumas de suas características, como *packing* e *unpacking*. A partir de agora, vamos criar uma série de pequenos programas para motivar e contextualizar melhor os recursos da linguagem Python.

Capítulo 4

Primeiro programa: download de dados da Copa 2014

4.1 Criando uma função para download na web

O nosso primeiro programa consiste basicamente em duas funções de download de dados. Uma delas leva em conta que o servidor responde o tamanho da base cujo download queremos fazer, e a outra trata quando o servidor não informa o tamanho. Esse download é feito via protocolo HTTP, usando recursos da própria biblioteca padrão.

Como sabemos que existem essas duas situações, podemos iniciar imaginando uma função que trata cada caso e depois combiná-las. Assim, teremos um programa que funciona em qualquer um dos casos descritos.

Download de arquivo de tamanho conhecido

Vamos considerar que o servidor nos informou o tamanho em bytes do arquivo do download no cabeçalho da requisição. Vamos ver o código, para então discutir os detalhes.

```
BUFF_SIZE = 1024
def download_length(response, output, length):
    times = length / BUFF_SIZE
    if length % BUFF_SIZE > 0:
       times += 1
    for time in range(times):
       output.write(response.read(BUFF_SIZE))
       print("Downloaded %d" % (((time * BUFF_SIZE)/length)*100))
```

O `response` representa uma resposta do servidor, sendo dela que leremos os bytes do arquivo de download. Como sabemos o tamanho, conseguimos saber quantas operações de `response.read()` vamos ter que realizar para ler tudo. Para cada leitura, realizamos um `output.write()` dos bytes lidos em um arquivo.

Nesse exemplo, exploramos todos os conceitos básicos que foram introduzidos até aqui: definição de função, atribuição de variáveis, operações aritméticas, chamadas de funções, formatação de strings e um loop com `range`.

Agora, vamos ver a outra função de download e, em seguida, a função que dispara uma delas, para criar o nosso primeiro programa completo.

Criando outra função para download na web

Como dito anteriormente, às vezes o servidor não responde o tamanho em bytes do arquivo que queremos. Nesses casos, realizamos leituras até que alguma não retorne nenhum byte. Basicamente, trocamos o loop com `for` e `range()` por um loop com `while`, visto que não conseguimos saber de antemão quando terminar. Quando a leitura não retorna nada, o comando `break` interrompe o `while`.

```
def download(response, output):
    total_downloaded = 0
    while True:
```

```
        data = response.read(BUFF_SIZE)
        total_downloaded += len(data)
        if not data:
            break
        output.write(data)
        print('Downloaded {bytes}'.format(bytes=total_downloaded))
```

Agora que já conhecemos as duas funções, veremos a função que escolhe chamar uma das duas.

4.2 Primeiro programa completo

O que inicialmente foi omitido: importações de módulos e a função `main`, que será chamada quando o programa for executado. Em muitas linguagens, o início dos arquivos contém tudo o que é importado de outros módulos.

```
# coding: utf-8
import io
import sys
import urllib.request as request
```

Intuitivamente, fica muito clara a importação dos módulos `io` e `sys`. A última linha é um `import` com alteração no *namespace* local: estamos importando `urllib.request` e colocando no nome local `request`. Se a parte `as request` fosse omitida, para usar o objeto `request`, teríamos que escrever `urllib.request` todas as vezes. Aqui a decisão é sempre de acordo com o contexto. Algumas vezes, queremos explicitar todos os momentos em que estamos usando uma função ou objeto que é de outro módulo, e outras queremos limpar o código de qualquer informação redundante. A escolha é sua!

Agora, vamos oficialmente aprender o que seria em Python a função `main` de outras linguagens, como C ou Java.

4.3 Definindo funções main

Em Python, quando chamamos o interpretador passando um arquivo `.py` como parâmetro, o padrão é que todas as linhas do arquivo sejam avalia-

das/executadas. O detalhe é que todo código de escopo global em um módulo também será executado quando for importado. O problema, então, é: como sabemos se um módulo foi aquele passado como parâmetro na linha de comando?

Basta testar o valor de uma variável global. Se o teste do `if` a seguir for verdadeiro, é porque esse módulo (que está fazendo esse teste) foi o chamado pela linha de comando. É bem comum algo como:

```
def main():
    print("Olá")

if __name__ == "__main__":
    main()
```

Nesse exemplo, `main` só é executada quando esse módulo é o utilizado na linha de comando. Quando ele é importado, a função `main` não é executada.

Voltando ao nosso exemplo, nossa `main`:

```
def main():
    response = request.urlopen(sys.argv[1])
    out_file = io.FileIO("saida.zip", mode="w")

    content_length = response.getheader('Content-Length')
    if content_length:
        length = int(content_length)
        download_length(response, out_file, length)
    else:
        download(response, out_file)

    response.close()
    out_file.close()
    print("Finished")

if __name__ == "__main__":
    main()
```

O que temos agora é um *script* que faz o download de um arquivo `ZIP`. De acordo com a resposta do servidor, ele opta por uma determinada estratégia para download do arquivo.

Utilizamos a função da biblioteca padrão `urllib.request.urlopen`, a classe `io.FileIO` para escrevermos o arquivo binário de saída, um comando `if` e algumas chamadas de funções. As funções contêm um `while`, um `for` e algumas outras operações já mencionadas. Note que com esses poucos conceitos já conseguimos construir um programa que realiza uma tarefa completa.

Veja o nosso primeiro programa, que faz o download dos arquivos de dados que usaremos daqui em diante:

```
# coding: utf-8

import io
import sys
import urllib.request as request

BUFF_SIZE = 1024


def download_length(response, output, length):
    times = length // BUFF_SIZE
    if length % BUFF_SIZE > 0:
        times += 1
    for time in range(times):
        output.write(response.read(BUFF_SIZE))
        print("Downloaded %d" % (((time * BUFF_SIZE)/length)*100))


def download(response, output):
    total_downloaded = 0
    while True:
        data = response.read(BUFF_SIZE)
        total_downloaded += len(data)
        if not data:
            break
        out_file.write(data)
        print('Downloaded {bytes}'.format(bytes=total_downloaded))
```

```
def main():
    response = request.urlopen(sys.argv[1])
    out_file = io.FileIO("saida.zip", mode="w")

    content_length = response.getheader('Content-Length')
    if content_length:
        length = int(content_length)
        download_length(response, out_file, length)
    else:
        download(response, out_file)

    response.close()
    out_file.close()
    print("Finished")


if __name__ == "__main__":
    main()
```

4.4 Realizando o download

Para facilitar, criamos um arquivo baseado nos dados originais, mas com algumas simplificações (sem todas as tabelas) e sem alguns dados "sujos" que poderiam trazer problemas ao rodar os exemplos.

Todos os exemplos funcionarão perfeitamente com o nosso conjunto de arquivos. Com os dados originais, alguns exemplos podem precisar de alterações, especialmente de tratamento de erro. A URL dos nossos arquivos é http://livropython.com.br/dados.zip.

Todos os exemplos do livro vão funcionar com a nossa versão dos arquivos de dados, já que os arquivos originais podem ter problemas de formatação difíceis de contornar.

Se você quiser os dados originais, você consegue encontrá-lo no próprio site do Portal da Transparência [1]. Se você concluir todas as etapas do livro, pode aventurar-se com os dados originais, que serão uma nova fonte de desafios, uma vez que diversos problemas poderão ser encontrados ao tentar explorar os dados originais.

Para obter o nosso arquivo de dados usando o nosso programa, crie um arquivo chamado `download_dados_copa.py` com o código-fonte anterior e execute a linha de comando:

```
$ python download_dados_copa.py
                        http://livropython.com.br/dados.zip
```

Agora, na raiz do projeto temos o arquivo `saida.zip`, que foi escrito pelo nosso programa a partir da resposta da requisição.

Essa tarefa é parte de uma aplicação maior, que automatiza o processo de extração dos dados da base de dados dos gastos com a copa de 2014 na base do Portal da Transparência [1].

Repare que, mesmo nesse programa relativamente complexo, usamos somente conceitos exibibos até agora.

Vamos continuar com as outras etapas deste processo.

4.5 Mais do básico: extraindo arquivos de dados e metadados

Antes de seguirmos para o próximo tópico, vamos nos exercitar um pouco mais com o que vimos até agora: funções, variáveis, condicionais, loops, condições de parada e o uso de alguns objetos importados de outros módulos da biblioteca padrão.

Em vez de utilizar um programa padrão para extrair o conteúdo do arquivo `ZIP` baixado, vamos criar um novo programa que usa o módulo `zipfile` para essa tarefa.

Note que faremos um tratamento de erro bem simples caso o arquivo passado como parâmetro não exista. Novamente, veremos apenas o uso de conceitos que já exibidos pelo menos uma vez, mesmo que sem muitos detalhes:

```
# coding: utf-8

import os
import zipfile
import sys
```

```
def main(path):
    if not os.path.exists(path):
        print("Arquivo {} não existe".format(path))
        sys.exit(-1)
    else:
        zfile = zipfile.ZipFile(path)
        zfile.extractall()
        print("Arquivos extraídos")


if __name__ == "__main__":
    main(sys.argv[1])
```

Vamos ver o script `extrai_zip.py` inteiro e analisar cada parte individualmente. Inicialmente, vemos a declaração do *encoding* – é uma boa prática sempre ter e, se você usa texto com acentos na sua documentação, é obrigatório definir quase sempre `utf-8`.

Importamos apenas 3 módulos: `sys`, `os` e `zipfile`. Com o módulo `sys`, pegamos o argumento da linha de comando – nesse caso, o caminho do arquivo baixado – e também podemos encerrar a execução do nosso programa com `sys.exit(return_code)`. Usamos o código de retorno `-1` porque o arquivo passado não existe. Para testar a existência do arquivo, usamos `os.path.exists(path)`. Essa função retorna `True` caso o caminho passado exista, e `False` caso não.

Existindo, vamos usar o módulo `zipfile` para criar um objeto da classe `ZipFile` que possui o método `extractall`. Este extrai todo o conteúdo do arquivo `zip` para o diretório de trabalho.

Com poucas linhas, e alguns conceitos e construtos iniciais, criamos dois programas: um deles faz o download de um arquivo `zip` de um servidor HTTP e o outro extrai o conteúdo do arquivo para o diretório corrente, para que os próximos programas possam usá-lo.

4.6 Conclusão

Assim, terminamos a primeira parte do nosso processo. Veja novamente que utilizamos apenas funções e strings, importamos alguns módulos e executamos alguns comandos. Ou seja, tudo o que vimos, mesmo que superficialmente, até agora.

É claro que muita coisa poderia ser feita de outra forma, mas a ideia desses scripts iniciais é ser o mais simples possível.

Nos próximos capítulos, falaremos mais de estruturas de dados e como vamos usá-las para alcançar nosso objetivo maior.

Capítulo 5

Estruturas de dados

5.1 Montando um modelo conceitual usando estruturas de dados

O objetivo do nosso programa é realizar consultas em parte dos dados dos gastos públicos da Copa 2014. Os arquivos disponibilizados dividem-se em dois tipos: metadados e dados. Os *dados* são como se fossem as linhas de um banco de dados, enquanto os *metadados* seriam a definição das colunas de uma tabela de um banco de dados, ou seja, nada mais são do que a descrição dos dados. Vamos utilizar os metadados como suporte para realizar consultas nos dados.

Depois das etapas anteriores, agora temos uma pasta que contém um arquivo de metadados para cada entidade do modelo de dados, e outra pasta com os dados em si. O que queremos é que nosso programa seja capaz de

abrir todos esses arquivos e interpretar o conteúdo de forma adequada. Vamos ver o conteúdo de um dos arquivos de metadados:

```
IdInstituicao    bigint    Identificador da instituição-PK.
IdTipoInstituicao bigint Identificador do tipo de instituição ...
NomInstituicao    varchar    Nome da instituição.
NumCnpj    varchar    Número do CNPJ.
```

Existem 3 "colunas" nesse arquivo: o nome do campo (por exemplo, `IdInstituicao`), o tipo do campo (por exemplo, `bigint`), e a descrição. Cada linha desse arquivo refere-se a uma informação de uma linha no arquivo de dados. Nesse exemplo, portanto, podemos esperar que o arquivo de dados da entidade *Instituição* tenha 4 valores separados pelo caractere `;` (ponto e vírgula).

Veja um trecho do arquivo de dados `Instituicao.csv`:

```
1;1;"Caixa Econômica Federal";"00360305000104"
2;1;"BNDES";"33657248000189"
8;3;"GOVERNO DO ESTADO DE MINAS GERAIS";"96313723000117"
12;5;"GOVERNO DO DISTRITO FEDERAL";"00394692000108"
105;6;"Concessionário";"72036339000159"
106;1;"BNB";"07237373000120"
107;2;"INFRAERO";"00352294000110"
```

Assim como descrito no arquivo de metadados de *Instituição*, cada linha contém 4 informações sobre a entidade à qual o arquivo faz referência. O nosso programa conseguirá ler os arquivos de 4 tipos distintos de entidades e, posteriormente, cruzar dados para permitir consultas mais completas e interessantes.

Para iniciar esse processo, primeiro precisamos de uma forma de encontrar os metadados de uma entidade dada a `string` com o seu nome. Para isso, usaremos um *Dicionário*. Esse dicionário de metadados, que será criado a partir da leitura dos arquivos, será um dos pontos centrais do nosso aplicativo.

5.2 Dicionários: fundação da linguagem

O *dicionário*, também muito conhecido como *mapa* ou *array associativo*, é um conceito abstrato de estrutura de dados [12], no qual temos N entradas associadas a uma ou mais chaves por entrada. Em Python, o famoso `dict` é uma das estruturas de dados mais utilizadas. Inclusive, muitos recursos da própria linguagem têm implementações que usam dicionários.

Nos `dicts`, as chaves devem ser imutáveis e os valores podem ser qualquer objeto. Ele é instanciado usando a sintaxe `{}`, ou a função `dict()`. Veja uns exemplos:

```
entidades = {
    'Instituicao': []
}
```

Também podemos usar:

```
entidades = dict(Instituicao=[])
```

Ou seja, há duas formas de realizar a mesma tarefa: usando a sintaxe literal, ou usando a função `dict()`. Também podemos adicionar itens ao dicionário após criado, como no exemplo a seguir:

```
entidades = dict()
entidades["Instituicao"] = []
```

Da mesma maneira que podemos atribuir valores para chaves, podemos também remover os valores usando o comando `del`, no elemento que queremos remover do dicionário.

```
>>> entidades = dict()
>>> entidades['Empreendimento'] = "EntidadeEmpreendimento"
>>> print(entidades)
{'Empreendimento': 'EntidadeEmpreendimento'}
>>> del entidades['Empreendimento']
>>> print(entidades['Empreendimento'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Empreendimento'
```

O que queremos agora é criar um dicionário, no qual as chaves são os nomes das entidades e os valores sejam listas. Essas listas contêm os campos com nome, tipo e descrição que compõe de cada atributo da entidade. No final dessa etapa, queremos ter algo parecido com o objeto a seguir:

```
entidades = {
    'Instituicao': [
        ('IdInstituicao', 'bigint',
         'Identificador da instituição-PK'),
        ('IdTipoInstituicao', 'bigint',
         'Id do tipo de instituição'),
        ('NomInstituicao', 'varchar', 'Nome da instituição'),
        ('NumCnpj', 'varchar', 'Número do CNPJ')
    ]
}
```

O que temos é um dicionário com chaves do tipo `string` e valores do tipo `list`. As listas, por sua vez, são do tipo `tuple` e contêm 3 elementos cada. Para cada um dos arquivos de metadados, teremos uma entrada no dicionário `entidades` com a lista de atributos que aquela entidade contém. A seguir, vamos ver como montar esse dicionário, que tem papel importante nos exemplos que virão a seguir no livro.

5.3 Montando o dicionário de metadados

Está com dúvidas sobre como montar esse dicionário? Não se preocupe.

Se você criou e rodou os dois programas completos que passamos até então, o que você tem disponível agora são 2 pastas. Uma dessas chama-se `meta-data` e contém 4 arquivos texto com os metadados que nos interessam.

Se você quiser, também podemos obter os dados que serão usados a partir daqui por aquele endereço com os arquivos do livro, http://livropython.com.br/dados.zip. Basta descompactar o arquivo `ZIP` em um diretório e rodar todos os exemplos nesse diretório.

Poderíamos, simplesmente, criar em código o dicionário que contém as mesmas informações que esses arquivos. Como isso pode ser um pouco trabalhoso, vamos criar um novo programa que lista os arquivos dessa pasta,

abre um de cada vez e monta as listas de atributos, baseando-se no conteúdo dos arquivos.

No parágrafo anterior, listamos três tarefas. Vamos ver exemplos separados de como realizar cada uma delas e depois criar um novo programa.

Listando arquivos de uma pasta

Para listar arquivos de uma pasta, podemos usar a função `listdir()` do módulo `os`. Ela retorna uma lista de arquivos, portanto, pode ser usada com o comando `for`. Veja o exemplo:

```
>>> import os
>>> for meta_file in os.listdir('data/meta-data'):
...     print(meta_file)
...
Empreendimento.txt
ExecucaoFinanceira.txt
Instituicao.txt
Licitacao.txt
```

Agora que listamos os arquivos, precisamos extrair a extensão `.txt` para obter o nome das entidades com que vamos trabalhar. Vimos a função `split()` do objeto `string`, que pode ser usada para essa tarefa. Ela também retorna uma lista, logo, para pegar a primeira parte, usamos o indexador `filename.split('.')[0]` para pegar o elemento na posição `0`. Veja o exemplo:

```
>>> def extract_entity_name(filename):
...     return filename.split('.')[0]
...
>>> extract_entity_name('Licitacao.txt')
'Licitacao'
```

Nesse código, obtemos o nome da entidade por meio do nome do arquivo texto associado. Por exemplo, `Licitacao.txt` transforma-se em `Licitacao`. Os nomes serão as chaves do dicionário e os valores serão uma lista de tuplas (que veremos logo a seguir) que descrevem os campos da entidade.

5.4 Adicionando e removendo elementos em uma lista

Com o que vimos, já poderíamos criar um dicionário com as chaves, mas sem a lista de atributos que queremos. Como os arquivos de metadados são padronizados, podemos criar um programa que funcione para todos que sigam os padrões definidos.

No início do capítulo, vimos que o conteúdo de um arquivo de metadados pode ser algo como:

```
IdInstituicao    bigint    Identificador da instituição-PK.
IdTipoInstituicao bigint Identificador do tipo de instituição ...
NomInstituicao    varchar    Nome da instituição.
NumCnpj    varchar    Número do CNPJ.
```

Cada linha tem 3 elementos separados pelo caractere `\t` (`tab`). Vamos ver como poderia ser uma função, que recebe um caminho para um arquivo de metadados e retorna uma lista de tuplas, como desejamos. Para adicionar uma tupla na lista de atributos, usaremos o método `append()`. O método `append` adiciona um novo elemento no final da lista.

```
>>> def read_meta_data(path):
...     data = open(path, "rt")
...     meta_data = []
...     for line in data:
...         line_data = line.split('\t')
...         meta_data.append((line_data[0],
...                           line_data[1],
...                           line_data[2]))
...     data.close()
...     return meta_data
...
>>> read_meta_data('data/meta-data/Instituicao.txt')
[('IdInstituicao', 'bigint', 'Identificador da instituição-PK.'),
('IdTipoInstituicao', 'bigint', 'Identificador do tipo de
instituição associada à instituição.'), ('NomInstituicao',
'varchar', 'Nome da instituição.'), ('NumCnpj', 'varchar',
'Número do CNPJ.')]
```

Poderíamos ter usado também o método `insert(i, o)`, em que `i` é a posição onde queremos adicionar um objeto, e `o` é o objeto que queremos adicionar.

Para remover objetos, poderíamos usar os métodos `remove(obj)` e `pop(i)`. O método `remove(obj)` remove `obj` da lista, já o `pop([i])` remove o elemento na posição `i` e o retorna. Se `i` não for especificado, o último elemento é removido e retornado.

Outras operações de listas são:

- `reverse()` – inverter a ordem dos elementos;
- `sort()` – ordenar por valor;
- `extend(lista)` – concatenar com outra lista;
- `index(elemento)` – descobrir a posição de um elemento;
- `clear()` – apagar todos os elementos da lista.

Veja alguns exemplos de uso:

```
>>> lista = [1, 2, 3, 4, 5]
>>> lista.append(6)
>>> lista
[1, 2, 3, 4, 5, 6]
>>> lista.insert(0, -1)
>>> lista
[-1, 1, 2, 3, 4, 5, 6]
>>> lista.remove(6)
>>> lista
[-1, 1, 2, 3, 4, 5]
>>> lista.pop(0)
-1
>>> lista
[1, 2, 3, 4, 5]
>>> lista.reverse()
>>> lista
[5, 4, 3, 2, 1]
>>> lista.sort()
```

```
>>> lista
[1, 2, 3, 4, 5]
>>> lista.index(5)
4
>>> lista_b = [6, 7, 8]
>>> lista.extend(lista_b)
>>> lista
[1, 2, 3, 4, 5, 6, 7, 8]
>>> lista.clear()
>>> lista
[]
```

5.5 Iterando dicionários: vendo valores e chaves

Em Python, temos uma forma simples de iterar pela dupla `chave -> valor` de um dicionário. Podemos usar `for key, value in data.items(): ...` e, a cada execução do bloco do `for`, teremos em `key` o objeto chave e em `value` o objeto valor daquela entrada. Isso acontece no final do programa:

```
import os


def extract_name(name):
    return name.split(".")[0]


def read_lines(filename):
    _file = open(os.path.join("data/meta-data", filename), "rt")
    data = _file.read().split("\n")
    _file.close()
    return data


def read_metadata(filename):
    metadata = []
    for column in read_lines(filename):
```

```
        if column:
            values = column.split('\t')
            nome = values[0]
            tipo = values[1]
            desc = values[2]
            metadata.append((nome, tipo, desc))
    return metadata


def main():
    meta = {}
    for meta_data_file in os.listdir("data/meta-data"):
        table_name = extract_name(meta_data_file)
        meta[table_name] = read_metadata(meta_data_file)

    for key, val in meta.items():
        print("Entidade {}".format(key))
        print("Attributes: ")
        for col in val:
            print("  {}: {}".format(col[1], col[0]))


if __name__ == "__main__":
    main()
```

Nesse programa, a chave é sempre uma `string` que corresponde ao nome da entidade, e o valor é a lista de atributos. Como os atributos estão em uma lista, sabemos que podemos usá-la no comando `for` e, assim, percorrer cada atributo.

Primeiro, listamos os arquivos da pasta de meta-dados. Para cada nome de arquivo, extraímos o nome da entidade. Depois, abrimos cada um dos arquivos de metadados e retiramos as informações das linhas do arquivo. Cada linha tem a informação de um atributo (ou coluna) da entidade (ou tabela) com nome, tipo e descrição.

No final, iteramos um dicionário com o comando `for`, e escrevemos na saída o nome da entidade e os detalhes dos atributos. Veja uma parte da saída que você verá ao executar esse código:

```
Entidade Instituicao
 Atributos:
  bigint: IdInstituicao
  bigint: IdTipoInstituicao
  varchar: NomInstituicao
  varchar: NumCnpj
```

5.6 Tuplas: sequências imutáveis

No exemplo anterior, cada entidade é representada por uma lista de tuplas. As tuplas, uma vez criadas, não podem ser mudadas: são imutáveis. No nosso caso, cada tupla a respeito de uma coluna sempre terá 3 elementos: nome, tipo e descrição. Além disso, inicialmente, não vamos considerar trocas de metadados, até porque os dados já estão prontos e não vão mudar.

Tuplas também são amplamente usadas na implementação de recursos da linguagem Python. Além de serem usadas como coleções imutáveis, também são muito utilizadas como agrupadores de elementos heterogêneos, como o `struct` de C.

Assim como listas e dicionários, tuplas são sequências. Então, podemos acessar elementos pelo índice, saber seu tamanho e se um elemento está dentro dela ou não. Vamos ver alguns exemplos:

```
>>> meta_dado = ('IdTipoAlerta', 'bigint',
                 'Identificador do tipo alerta-PK.')
>>> 'IdTipoAlerta' in meta_dado
True
>>> len(meta_dado)
3
>>> meta_dado[0]
'IdTipoAlerta'
>>> impostos[0] = 'OutroValor'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Agora, já compreendemos o emprego do *Dicionário* e da *Tupla* no nosso programa. O próximo objetivo é detectar relacionamentos automaticamente, analisando os metadados que representamos.

5.7 Explorando os dados que coletamos

Nesse momento, o que temos, precisamente, é um dicionário de listas de tuplas. Apesar de parecer complexo, o nosso modelo é bem simples. Vamos aprender mais alguns métodos para explorar a nossa estrutura e conhecer melhor o que montamos.

Primeiro, vamos descobrir que entidades compõem essa base de dados. Uma forma simples de ver isso seria adaptar nosso penúltimo código para imprimir somente o nome das entidades:

```
import os

def main():
    meta = {}
    for meta_data_file in os.listdir("data/meta-data"):
        table_name = meta_data_file.split('.')[0]
        print(table_name)

if __name__ == "__main__":
    main()
```

E a saída seria:

```
Empreendimento
ExecucaoFinanceira
Instituicao
Licitacao
```

Como já sabemos as entidades e os atributos, podemos buscar por relações entre entidades automaticamente. No contexto dos nossos dados, uma relação é quando existe, em alguma entidade, um atributo que aponta para um elemento de outra entidade.

Sabemos que cada entidade tem um atributo identificador. Portanto, vamos procurar em cada entidade um atributo identificador de outra entidade. Quando acharmos algum, encontramos uma relação.

O penúltimo exemplo cria um dicionário de entidades, no qual os valores são listas de tuplas. O primeiro elemento de cada lista é a chave identificadora da entidade.

Para descobrir as relações, primeiro temos que achar as chaves identificadoras de cada entidade. Vamos, então, criar um dicionário que as chaves são as `strings` que representam os nomes, e os valores são o nome da entidade. Vamos chamá-lo de dicionário `identificador para nome da entidade`.

Com ele, poderemos verificar se o seu nome está no dicionário, a cada atributo encontrado para uma entidade. Se estiver, significa que essa entidade, cujos atributos são o que estamos iterando, tem uma referência para outra. A entidade é o valor do dicionário de identificador para nome da entidade.

```
import os


def extract_name(name):
    return name.split(".")[0]


def read_lines(filename):
    _file = open(os.path.join("data/meta-data", filename), "rt")
    data = _file.read().split("\n")
    _file.close()
    return data


def read_metadata(filename):
    metadata = []
    for column in read_lines(filename):
        if column:
            values = column.split('\t')
            nome = values[0]
            tipo = values[1]
            desc = values[2]
            metadata.append((nome, tipo, desc))
    return metadata


def main():
    # dicionário nome entidade -> atributos
```

```
    meta = {}

    # dicionário identificador -> nome entidade
    keys = {}

    for meta_data_file in os.listdir("data/meta-data"):
        table_name = extract_name(meta_data_file)
        attributes = read_metadata(meta_data_file)
        identifier = attributes[0][0]

        meta[table_name] = attributes
        keys[identifier] = table_name

    for key, val in meta.items():
        for col in val:
            if col[0] in keys:
                if not col[0] == meta[key][0][0]:
                    print("Entiade {} -> {}".format(key, col[0]))


if __name__ == "__main__":
    main()
```

No final do programa anterior, iteramos todos os atributos de todas as entidades. Quando um atributo de uma entidade qualquer tem o mesmo nome de um identificador de outra entidade, nós encontramos uma referência. Essa referência é muito semelhante ao conceito de chave estrangeira de um SGBD (Sistema de Gerenciamento de Banco de Dados, do inglês *Data Base Management System*). Uma coluna desse tipo nos permite ligações entre dados de entidades distintas. Como o nosso objetivo maior é explorar os dados, damos um passo à frente com essa funcionalidade.

Se você rodar o programa anterior, deverá ver:

```
Entiade ExecucaoFinanceira aponta para IdEmpreendimento
Entiade ExecucaoFinanceira aponta para IdLicitacao
Entiade Empreendimento aponta para IdInstituicao
Entiade Licitacao aponta para IdEmpreendimento
```

Isso significa que podemos montar um modelo conceitual do nosso pe-

queno universo de dados.

Resultado final do uso das estruturas de dados

Ao final desse programa, usamos todas as 3 fundamentais estruturas de dados Python: tuplas, listas e dicionários. Cada uma delas serviu a um propósito específico: os dicionários foram usados para obtermos objetos de nosso interesse quando em posse de uma chave; a lista foi usada na criação da lista de atributos de uma entidade; e as tuplas foram usadas para representar as 3 informações de um atributo (nome, tipo e descrição).

Repare que ainda estamos usando recursos bem fundamentais da linguagem. Basicamente, nosso programa mais complexo, até o momento, lê um arquivo, itera suas linhas, aplica algumas transformações nos dados e monta um conjunto de estruturas de dados que nos permitem observar informações sobre os metadados.

Com essas estruturas em mãos, podemos:

- Listar as entidades;
- Ver os atributos das entidades;
- Identificar relacionamentos entre entidades.

Vamos criar um exemplo interativo para explorar os metadados.

5.8 Exemplo final usando estruturas de dados para explorar os metadados

Aqui, vamos unir boa parte do que vimos até agora:

- Strings e números;
- Definição de função e chamadas a funções;
- Loops com `for`;
- Estruturas de dados.

```
import os


def extract_name(name):
    return name.split(".")[0]


def read_lines(filename):
    _file = open(os.path.join("data/meta-data", filename), "rt")
    data = _file.read().split("\n")
    _file.close()
    return data


def read_metadata(filename):
    metadata = []
    for column in read_lines(filename):
        if column:
            metadata.append(tuple(column.split('\t')[:3]))
    return metadata


def prompt():
    print("\nO que deseja ver?")
    print("(l) Listar entidades")
    print("(d) Exibir atributos de uma entidade")
    print("(r) Exibir referências de uma entidade")
    print("(s) Sair do programa")
    return input('')


def main():
    # dicionário nome entidade -> atributos
    meta = {}

    # dicionário identificador -> nome entidade
    keys = {}

    # dicionário de relacionamentos
```

```
    relationships = {}

    for meta_data_file in os.listdir("data/meta-data"):
        table_name = extract_name(meta_data_file)
        attributes = read_metadata(meta_data_file)
        identifier = attributes[0][0]

        meta[table_name] = attributes
        keys[identifier] = table_name

    for key, val in meta.items():
        for col in val:
            if col[0] in keys:
                if not col[0] == meta[key][0][0]:
                    relationships[key] = keys[col[0]]

    opcao = prompt()
    while opcao != 's':
        if opcao == 'l':
            for entity_name in meta.keys():
                print(entity_name)
        elif opcao == 'd':
            entity_name = input('Nome da entidade: ')
            for col in meta[entity_name]:
                print(col)
        elif opcao == 'r':
            entity_name = input('Nome da entidade: ')
            other_entity = relationships[entity_name]
            print(other_entity)
        else:
            print("Inexistente\n")
        opcao = prompt()


if __name__ == "__main__":
    main()
```

Esse programa nos dá a possibilidade de explorar os metadados de forma interativa. O que queremos daqui para a frente é usar os metadados para

explorar os dados em si.

Uma novidade nesse código apresentado foi o uso do método `keys()` do dicionário. Em Python, podemos iterar um dicionário de 3 formas:

- Por chave e valor, usando `for key, val in my_dict.items()`;
- Por chaves, usando `for key in my_dict.keys()`;
- Por valores, usando `for val in my_dict.values()`.

Veja alguns exemplos:

```
>>> meu_dict = {'Empreendimento': [('IdEmpreendimento', 'biging',
...             '...')], 'Licitacao': [('IdLicitacao', 'bigint',
...             '...')]}
>>> for (name, attributes) in meu_dict.items():
...     print("Nome {} com {} atributo(s)".format(name,
                                            len(attributes)))
...
Nome Licitacao com 1 atributo(s)
Nome Empreendimento com 1 atributo(s)
>>> for name in meu_dict.keys():
...     print(name)
...
Licitacao
Empreendimento
>>> for attributes in meu_dict.values():
...     print(attributes)
...
[('IdLicitacao', 'bigint', '...')]
[('IdEmpreendimento', 'biging', '...')]
```

Uma deficiência do código que criamos até agora é que os conceitos presentes nas explicações não têm uma representação direta no código. No universo da programação, existem diversos paradigmas, como *Orientação a Objetos* e *Funcional*, que influenciariam a melhor organizar o código que temos até então, ou repensar algumas de nossas soluções até o momento. Python, como mencionado no início deste livro, é uma linguagem *multiparadigma*, ou seja, você consegue explorar recursos de vários paradigmas nela.

Na sequência, exploraremos um pouco do paradigma da *Orientação a Objetos* para explicar o mecanismo de classes e objetos em Python. Vamos criar classes que façam sentido para o nosso aplicativo, para melhorar a implementação, isolar algumas coisas de mais *baixo nível* e criar uma camada de abstração que permitirá que você faça mais coisas no projeto.

5.9 Estruturas de dados são importantes?

Certamente!

Um dos temas mais importantes na Ciência da Computação são as estruturas de dados. Elas oferecem meios eficientes de realizarmos tarefas em conjuntos de dados. Este capítulo foi uma introdução as 3 estruturas de dados mais comuns em Python.

5.10 Conclusão

Neste capítulo, apresentamos o que são tuplas e dicionários. Junto com as listas, são as principais estruturas de dados da linguagem. No dia a dia, vamos utilizá-las sempre. Mostramos um pouco da motivação de usar as estruturas, dadas as características individuais de cada uma.

Vimos como utilizar uma combinação de estruturas para representar as informações que queremos em nosso programa. Para aprender todos os detalhes, a recomendação oficial é o lugar mais adequado.

A documentação completa dessas estruturas encontra-se em https://docs.python.org/3/tutorial/datastructures.html.

No próximo capítulo, evoluiremos o design interno do nosso aplicativo, seguindo um modelo orientado a objetos.

Capítulo 6

Classes e objetos pythônicos

Já temos um programa que, a partir dos nossos dados, monta uma estrutura que representa todas as entidades e relacionamentos de uma parte das informações da base de dados do Portal da Transparência [1]. O que ainda não temos é uma forma de consultar os dados.

Para isso, vamos realizar duas tarefas: uma é garantir que os dados nos arquivos de dados sejam compatíveis com a estrutura definida no arquivo de metadados; e a outra é criar os componentes que vão compor a consulta de dados.

O que queremos realizar é **apenas uma pequena parte** do que um banco de dados faz, os famosos *SGBDs*. Vamos modelar objetos que mapeiem conceitos de um banco de dados para unidades de código. O modelo de *Orientação a Objetos* é um paradigma muito forte na comunidade Python. Ainda que não seja seu paradigma preferido, é importante entender como Python permite aplicá-lo.

6.1 Expressando o domínio em classes

Quando falamos em *domínio*, estamos falando dos conceitos que nosso programa opera. Os dados que vamos explorar são dados de gastos públicos, mas poderiam ser sobre qualquer outro tipo de dado tabulado. Seguindo esse raciocínio, vemos que o domínio do nosso programa são *dados tabulados*.

Com uma classe, podemos expressar o conceito de **tabela** e juntar a ela todos os atributos e comportamentos esperados. Classes servem tanto para organização de código quanto forma de expressar um conceito do domínio. Nesse caso, vamos juntar os dados e os metadados em uma coisa única que permita a validação e leitura dos dados.

O que o nosso programa faz até agora consiste em 3 funcionalidades: fazer o download da base de dados; extrair o arquivo baixado e ler os metadados; montar a relação de dependências entre as entidades presentes. O download e a extração dos dados continuarão da mesma forma. Porém, tudo relacionado aos dados e metadados será feito por objetos que vamos criar a partir de agora.

A ideia é criar um minibanco de dados em memória que permita inserir linhas, validando os dados, de acordo com as definições dos metadados, e permita consultas com filtros. Essa tarefa pode parecer complicada, mas, aos poucos, vamos construindo componentes que tornarão isso possível, de uma forma bem elegante e *pythônica*.

Agora, vamos passar a modelar o nosso domínio com classes e entender como Python nos permite isso.

6.2 Definindo classes: primeiro passo

Python, assim como C# ou Java, possui classes, mas cada uma dessas linguagens tem peculiaridades em suas implementações.

Uma classe permite que *estado* e *comportamento* façam parte da mesma unidade de código. Quando falamos em *estado*, falamos dos *atributos* de uma classe; já quando falamos em *comportamento*, falamos de seus *métodos*. Quando criamos uma classe, ela deve ter um objetivo bem claro e sua interface tem de ser coerente com o que ela representa e disponibiliza.

Para termos um minibanco de dados em memória, precisamos de alguma forma de manipular tabelas de dados. O nosso primeiro objetivo é declarar

uma classe e definir valores para alguns atributos. No caso da nossa classe DataTable, os atributos iniciais serão: name (string), columns (lista) e data (lista). Esses atributos são exatamente os que temos nos arquivos de metadados, e os dados começam vazios. Cada entidade da base de dados terá uma instância de DataTable associada.

Em termos de sintaxe, definir uma classe em Python é extremamente fácil. Vamos ver como definir uma classe sem nada, instanciar e aplicar a função type() para ver o resultado:

```
>>> class DataTable:
...     pass
...
>>> table = DataTable()
>>> type(table)
<class '__main__.DataTable'>
```

O que vemos é <class '__main__.DataTable'>. Isso acontece pois estamos rodando em um console e, nesse caso, o módulo corrente é __main__. Agora temos a classe DataTable. Uma característica interessante é que, em Python, as instâncias das classes podem ser modificadas em tempo de execução. Um exemplo disso é que podemos adicionar atributos em tempo de execução. Veja o exemplo:

```
>>> class DataTable:
...     pass
...
>>> table = DataTable()
>>> table.name = 'Alerta'
>>> table.columns = ['IdProjeto', 'DescProjeto']
>>> table.data = []
```

O problema desse código é que não garantimos que todas as instâncias de DataTable tenham os atributos name e column. Isso precisa ser feito de alguma forma. Como muitas vezes queremos criar objetos já com determinadas configurações iniciais, é melhor ter isso de uma forma padronizada. Existe uma forma padronizada de definir os atributos de um objeto, geralmente, no método *construtor*. A seguir, veremos como criá-lo.

6.3 Criando objetos: métodos construtores

Em Python, alguns nomes de métodos estão reservados para o uso da própria linguagem. Um desses métodos é o que representa o *construtor*, sendo seu nome `__init__()`. Outra característica é que, pelo *The Zen of Python*, **explícito é melhor que implícito**. Logo, o primeiro parâmetro do construtor – assim como em todos métodos de instância – é sempre a própria instância. No nosso caso, é a que está sendo criada. Por **convenção**, esse argumento tem o nome de `self`. Veja no exemplo a definição de duas classes:

```
class DataTable:
    def __init__(self, name):
        self._name = name
        self._columns = []
        self._data = []

class Column:
    def __init__(self, name, kind, description):
        self._name = name
        self._kind = kind
        self._description = description
```

Note que, em nenhum momento, usamos palavras como `public` ou `private`, nem declaramos previamente quais seriam os atributos da classe.

Quando uma classe é criada, todos os atributos serão inicializados conforme o código do construtor. Na prática, quando executamos `DataTable("Empreendimento")`, a função `__init__()` da classe `DataTable` é executada passando os parâmetros da chamada. Assim, garantimos que todas instâncias de `DataTable` e `Column` tenham os atributos de que precisamos.

Encapsulamento sem private ou algo semelhante

Embora o encapsulamento não possa ser forçado – usando algo como a palavra reservada `private`, em Java –, ele pode ser implementado com *getters*, *setters* e até mesmo com o *decorator* `@property`. Uma convenção muito respeitada é representar atributos que seriam privados com um caractere _ (*underline*) antes, como em `self._data`.

Dessa forma, se quiséssemos expor o conteúdo de `_data`, poderíamos ter um *getter*:

```
>>> class DataTable:
...     def __init__(self, name):
...         self._name = name
...         self._columns = []
...         self._data = []
...     def getData(self):
...         return self._data
...
>>>
>>> table = DataTable("TabelaTeste")
>>> print(table.getData())
[]
```

No exemplo anterior, teríamos que usar `getData()` para acessar o atributo `_data`.

Ou poderíamos usar um *decorator*:

```
>>> class DataTable:
...     def __init__(self, name):
...         self._name = name
...         self._columns = []
...         self._data = []
...     @property
...     def data(self):
...         return self._data
...
>>> table = DataTable("TabelaTeste")
>>> print(table.data)
[]
```

A vantagem do *decorator* `@property` é que poderíamos acessar o atributo usando apenas `data`, como se `DataTable` tivesse um atributo com esse nome.

6.4 Classes documentadas: docstrings

Além do código da classe, ela deve ter preferencialmente alguma documentação. Assim como em outras linguagens, a documentação pode ser expressada junto ao código. Mas Python, por sua vez, também disponibiliza a documentação em tempo de execução.

Quando criamos uma classe, é importante expressar quais os atributos e para que servem, e falar dos métodos e o que mais for necessário para que os usuários do seu código a usem.

Uma ótima forma de fazer isso é com *docstrings*. Essas strings se tornam o atributo `__doc__` das classes. Veja o exemplo:

```
class DataTable:
    """Representa uma Tabela de dados.

       Essa classe representa uma tabela de dados do portal
       da transparência. Deve ser capaz de validar linhas
       inseridas de acordo com as colunas que possui. As
       linhas inseridas ficam registradas dentro dela.

       Attributes:
           name: Nome da tabela
           columns: [Lista de colunas]
           data: [Lista de dados]
    """
    def __init__(self, name):
        """Construtor

            Args:
                name: Nome da Tabela
        """
        self._name = name
        self._columns = []
        self.data = []

class Column:
    """Representa uma coluna em um DataTable
```

```
    Essa classe contém as informações de uma coluna
    e deve validar um dado de acordo com o tipo de
    dado configurado no construtor.

    Attributes:
        name: Nome da Coluna
        king: Tipo do Dado (varchar, bigint, numeric)
        description: Descrição da coluna
"""
def __init__(self, name, kind, description=""):
    """Construtor

        Args:
            name: Nome da Coluna
            kind: Tipo do dado (varchar, bigint, numeric)
            description: Descrição da coluna
    """
    self._name = name
    self._kind = kind
    self._description = description
```

Salve no arquivo `domain.py` e depois rode o código a seguir, no console Python:

```
>>> from domain import *
>>> print(DataTable.__doc__)
Representa uma Tabela de dados.

    Essa classe representa uma tabela de dados do portal
    da transparência. Deve ser capaz de validar linhas
    inseridas de acordo com as colunas que possui. As
    linhas inseridas ficam registradas dentro dela.

    Attributes:
        name: Nome da tabela
        columns: [Lista de colunas]
        data: [Lista de dados]

>>> print(Column.__init__.__doc__)
```

```
Construtor

        Args:
            name: Nome da Coluna
            kind: Tipo do dado (varchar, bigint, numeric)
            description: Descrição da coluna
```

Agora que temos uma classe com método *construtor* e uma explicação da sua existência, mesmo que seja inicial, podemos seguir adicionando comportamentos nela.

6.5 Métodos: adicionando comportamentos ao objeto

Vamos adicionar um método à nossa classe:

```
class DataTable:
    def __init__(self, name):
        self._name = name
        self._columns = []
        self._data = []

    def add_column(self, name, kind, description):
        column = Column(name, kind, description)
        self._columns.append(column)
        return column
```

O método `add_column` adiciona uma instância de `Column` em uma lista, em `DataTable`. Conceitualmente, podemos pensar que uma tabela tem uma coleção de colunas, e essa coleção é representada pela lista `self._columns`.

De forma semelhante, poderíamos ter um método `add_references()` para adicionar quais tabelas são apontadas pela instância, e `add_referenced()` onde adicionamos as tabelas que referenciam a própria instância. Para guardar essas informações, precisamos de dois novos atributos em `DataTable`: `_references`, que são quais tabelas ela aponta; e `_referenced`, que são as tabelas que apontam para ela.

Com as informações que sabemos extrair, já podemos dizer quais tabelas existem e quem aponta para quem, especificando as colunas dessas referências. O que não temos é uma classe que represente essa noção de *relacionamento*.

Esse *relacionamento* existe quando temos uma coluna de uma tabela que é uma chave primária de outra. O que determina um relacionamento é: uma coluna (onde ele existe), uma tabela de onde sai e uma tabela aonde ele chega. Junto a isso, vamos também dar um nome para esse relacionamento.

Algumas *docstrings* serão omitidas por conveniência. Vamos adicionar a classe `Relationship`:

```
class Relationship:
    """Classe que representa um relacionamento entre DataTables

       Essa classe tem todas as informações que identificam um
       relacionamento entre tabelas. Em qual coluna ele existe,
       de onde vem e pra onde vai.
    """
    def __init__(self, name, _from, to, on):
        """Construtor

           Args:
               name: Nome
               from: Tabela de onde sai
               to: Tabela pra onde vai
               on: instância de coluna onde existe
        """
        self._name = name
        self._from = _from
        self._to = to
        self._on = on

class DataTable:
    def __init__(self, name):
        self._name = name
        self._columns = []
        self._references = []
        self._referenced = []
```

```
        self._data = []

    def add_column(self, name, kind, description=""):
        column = Column(name, kind, description=description)
        self._columns.append(column)
        return column

    def add_references(self, name, to, on):
        """Cria uma referencia dessa tabela para uma outra tabela

           Args:
              name: nome da relação
              to: instância da tabela apontada
              on: instância coluna em que existe a relação
        """
        relationship = Relationship(name, self, to, on)
        self._references.append(relationship)

    def add_referenced(self, name, by, on):
        """Cria uma referência para outra tabela que aponta para
        essa.

           Args:
              name: nome da relação
              by: instância da tabela que aponta para essa
              on: instância coluna em que existe a relação
        """
        relationship = Relationship(name, by, self, on)
        self._referenced.append(relationship)
```

Agora temos 4 métodos na classe `DataTable`: o construtor e três métodos de *domínio*. Dois desses métodos são muito semelhantes, então vamos refatorar um pouco mais.

Coloque o código anterior, deste capítulo, no arquivo `domain.py`. Em um outro arquivo, ou mesmo no terminal, você pode usar essas classes. Veja o código a seguir:

```
>>> from domain import DataTable
>>> table_empreedimento = DataTable("Empreendimento")
```

```
>>> col_id = table_empreedimento.add_column('IdEmpreendimento',
                                             'bigint')
>>> col_aditivo = table_empreedimento.add_column('IdAditivo',
                                                  'bigint')
>>> col_alerta = table_empreedimento.add_column('IdAlerta',
                                                 'bigint')
>>> table_aditivo = DataTable("Aditivo")
>>> col_id = table_aditivo.add_column('IdAditivo', 'bigint')
>>> col_emp_id =
    table_empreedimento.add_column('IdEmpreendimento', 'bigint')
>>> table_empreedimento.add_references("IdAditivo",
                                     table_aditivo, col_aditivo)
>>> table_aditivo.add_referenced("IdEmpreendimento",
                                table_empreedimento, col_emp_id)
```

O nosso código agora representa 2 tabelas, e `Empreendimento` referencia e é referenciada por `Aditivo`. O que faltou no exemplo é que, na hora em que nosso programa estiver lendo os metadados, ele precisa detectar automaticamente os relacionamentos. Para isso, precisamos dar um passo adiante.

Aos poucos, nosso modelo de classes ganha mais comportamentos, mas ainda estamos um pouco longe de ter algo mais completo. Vamos resolver um problema de cada vez. O próximo é identificar os relacionamentos procurando pelas chaves primárias.

Nos arquivos de metadados, um dos tipos de coluna tem o tipo `PK`. Isso representa que aquela coluna é a chave primária (*primary key*) da tabela correspondente. Sempre que achamos uma coluna com nome igual ao de alguma chave primária é porque achamos uma referência entre as tabelas. Logo, podemos pensar na chave primária como um tipo especial de coluna.

6.6 Herança simples em Python

Podemos dizer que um *relacionamento*, nos nossos dados, existe quando uma coluna de uma tabela é a chave primária de outra. A coluna `IdAlerta` na tabela `Empreendimento` diz que `Empreendimento` referencia `Alerta`. Essa coluna é chave primária em `Alerta`. Parte da nossa lógica é: sempre que acharmos uma chave primária, ela será guardada em um dicionário pelo

nome, e ao iterarmos por outras, consultaremos esse dicionário. Caso exista uma chave primária, é porque temos uma relação.

Com herança, conseguimos criar um novo tipo, `PrimaryKey` que herda de `Column`, mantendo os seus atributos e modificando o valor de `_is_pk`.

```
class PrimaryKey(Column):
    def __init__(self, table, name, kind, description=None):
        super().__init__(name, kind, description=description)
        self._is_pk = True
```

A referência da superclasse fica na mesma linha do nome da classe, entre parênteses. Além disso, repare que foi utilizada a função `super()` para acessar o construtor da superclasse. Podemos implementar um método novo em `Column` e usar em instâncias de `PrimaryKey`.

```
class Column:
    def __init__(self, name, kind, description=""):
        self._name = name
        self._kind = kind
        self._description = description
        self._is_pk = False

    def __str__(self):
        _str = "Col: {} : {} {}".format(self._name,
                                        self._kind,
                                        self._description)

        return _str

class PrimaryKey(Column):
    def __init__(self, table, name, kind, description=""):
        super().__init__(name, kind, description=description)
        self._is_pk = True
```

Novamente, salve no arquivo `domain.py` e, no terminal, execute os seguintes comandos:

```
>>> from domain import *
>>> table = DataTable("Empreendimento")
>>> print(Column('IdEmpreendimento', 'bigint'))
```

```
Col: IdEmpreendimento : bigint
>>> print(PrimaryKey(table, 'IdEmpreendimento', 'bigint'))
Col: IdEmpreendimento : bigint
```

Aqui, vemos que o método especial `__str__()` foi implementado e herdado pela instância de `PrimaryKey`. Esse método também é um dos métodos especiais e tem a função semelhante ao `toString()` do Java, por exemplo, retornando uma representação do objeto em forma de `String`.

Esse método pode ser sobrescrito em `PrimaryKey`. Basta declará-lo nela mesma:

```
class PrimaryKey(Column):
    def __init__(self, table, name, kind, description=""):
        super().__init__(name, kind, description=description)
        self._is_pk = True

    def __str__(self):
        _str = "Col: {} : {} {}".format(self._name,
                                        self._kind,
                                        self._description)

        return "{} - {}".format('PK', _str)

class Column:
    def __init__(self, name, kind, description=""):
        self._name = name
        self._kind = kind
        self._description = description
        self._is_pk = False

    def __str__(self):
        _str = "Col: {} : {} {}".format(self._name,
                                        self._kind,
                                        self._description)
        return _str
```

Novamente, salve em `domain.py`. Execute e veja o resultado:

```
>>> from domain import *
>>> table = DataTable("Empreendimento")
```

```
>>> print(Column('IdEmpreendimento', 'bigint'))
Col: IdEmpreendimento : bigint
>>> print(PrimaryKey(table, 'IdEmpreendimento', 'bigint'))
PK - Col: IdEmpreendimento : bigint
```

O que ficou ruim é que a classe `DataTable` não implementa `__str__()`, mas isso pode ser facilmente resolvido depois.

Verificando se um objeto é instância de uma classe

Um dos objetivos do programa é garantir que os dados sejam validados antes do modelo ser montado em memória. Para isso, podemos colocar uma função de validação na classe `Column` e usá-la em outras partes do código, provavelmente onde as informações dos arquivos são lidas.

Para implementar a validação, vamos usar a função `isinstance(value, type)` e uma classe chamada `Decimal`. Felizmente, os tipos definidos na base do Copa Transparente (dados do Portal da Transparência [1]) podem ser mapeados diretamente para tipos em Python. Isso permite que algumas verificações sejam feitas com a função embutida `isinstance(value, type)`. No nosso código, o `'bigint'` é mapeado para um inteiro, `'numeric'` para `Decimal` e `'varchar'` para `string`s.

```
from decimal import Decimal

class Column:
    def __init__(self, name, kind, description=""):
        self._name = name
        self._kind = kind
        self._description = description

    def __str__(self):
        return "Col: {} : {} {}".format(self._name,
                                        self._kind,
                                        self._description)

    def validate(self, data):
        if self._kind == 'bigint':
```

```
            if isinstance(data, int):
                return True
            return False
        elif self._kind == 'varchar':
            if isinstance(data, str):
                return True
            return False
        elif self._kind == 'numeric':
            try:
                val = Decimal(data)
            except:
                return False
            return True
```

Vamos testar a função `validate()` no console:

```
>>> from domain import *
>>>
>>> c1 = Column("IdEmpreendimento", "bigint")
>>> c1.validate(100)
True
>>> not c1.validate(10.1)
True
>>> not c1.validate("Texto")
True
>>> c1 = Column("DescEmpreendimento", "varchar")
>>> c1.validate("Contrato")
True
>>> not c1.validate(10.1)
True
>>> not c1.validate(10)
True
>>> c1 = Column("ValTotalPrevisto", "numeric")
>>> c1.validate(10.1)
True
>>> c1.validate(10)
True
>>> not c1.validate("Texto")
True
```

O que vemos no resultado é que o valor `100` é um `'bigint'` válido, `10.1` é um `'numeric'` válido e `'Contrato'` é um `'varchar'` válido. Todas as outras situações não são válidas e o retorno é invertido pelo `not`, imprimindo `True`.

Com esse método, toda vez que uma linha do arquivo de dados for lida, poderemos validar o seu conteúdo com as regras das colunas que criamos com os metadados. Se uma linha tiver todas informações válidas, ela poderá ser inserida na lista de dados da tabela.

6.7 Atributos de classes: compartilhados entre todas instâncias

Como em outras linguagens, os atributos de classe são compartilhados entre todas as instâncias de uma classe. A mudança desejada é tornar o método de instância `validate()` um método estático ou de classe – que são muito semelhantes. A primeira etapa para fazer isso é saber como criar um atributo de classe.

Veja o exemplo:

```
class Column:
    def _validate(self, kind, data):
        if kind == 'bigint':
            if isinstance(data, int):
                return True
            return False
        elif kind == 'varchar':
            if isinstance(data, str):
                return True
            return False
        elif kind == 'numeric':
            try:
                val = Decimal(data)
            except:
                return False
            return True

    validate = _validate
```

Na última linha, declaramos um atributo de classe chamado `validate` e atribuímos a ele o método `_validate`. O problema é que isso não funciona dessa forma. Veja as execuções a seguir:

```
>>> class Column:
...     def _validate(self, kind, data):
...         if kind == 'bigint':
...             if isinstance(data, int):
...                 return True
...             return False
...         elif kind == 'varchar':
...             if isinstance(data, str):
...                 return True
...             return False
...         elif kind == 'numeric':
...             try:
...                 val = Decimal(data)
...             except:
...                 return False
...             return True
...     validate = _validate
...
>>> Column.validate('bigint', 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: _validate() missing 1 required positional argument:
'data'
```

Como vimos, para métodos de instância, o primeiro parâmetro é a própria instância sendo referenciada. No nosso caso, não existe instância, já que declaramos um atributo de classe e vamos referenciá-lo pela classe. Vamos ver como fazer funcionar o código anterior.

6.8 Métodos estáticos e de classe: usando mais as classes

Podemos transformar o método de validação em um método estático. Em Python, os métodos estáticos de classes são os que podem ser chamados

usando uma referência para uma classe `Column.validate()`, ou instância `col_instance.validate()`. O mais comum é usar a referência da classe.

A diferença entre o método estático e o de classe é que, neste, o primeiro argumento é a instância da classe. Como no nosso caso não precisamos da instância da classe, usamos um método estático. Em ambos, o parâmetro `self` dos métodos de instância não é passado.

Para definir um método estático, usamos uma função chamada `staticmethod()`, que recebe como argumento um método. E declaramos o resultado da chamada de `staticmethod` como um atributo de classe, da nossa classe.

```
from decimal import Decimal

class Column:
    def __init__(self, name, kind, description=""):
        self._name = name
        self._kind = kind
        self._description = description

    def _validate(kind, data):
        if kind == 'bigint':
            if isinstance(data, int):
                return True
            return False
        elif kind == 'varchar':
            if isinstance(data, str):
                return True
            return False
        elif kind == 'numeric':
            try:
                val = Decimal(data)
            except:
                return False
            return True

    validate = staticmethod(_validate)
```

O segredo aqui é que `_validate`, agora, não foi declarado com a assi-

natura de método de instância, que leva em consideração que `self` é sempre o primeiro argumento. O que queremos é um método genérico, que não precise de uma instância criada e que, dado o tipo e o valor, retorna falso ou verdadeiro, se o valor for correto para o tipo.

Para que essa função possa ser acessada pela referência da classe `Column`, precisamos informar que o método `_validate` é estático e que é acessado no nome `validate`, como usamos na declaração da última linha do exemplo.

Novamente, atualize o arquivo `domain.py` e execute no console Python:

```
>>> Column.validate('bigint', 100)
True
>>> Column.validate('numeric', 10.1)
True
>>> Column.validate('varchar', 'Texto')
True
```

Para transformar em método de classe, usamos a função `classmethod`. A diferença para `staticmethod` é que, em `classmethod`, o primeiro argumento da assinatura é a própria classe `Column`. Veja em código:

```
class Column:
    def _validate(cls, kind, data):
        if kind == 'bigint':
            if isinstance(data, int):
                return True
            return False
        elif kind == 'varchar':
            if isinstance(data, str):
                return True
            return False
        elif kind == 'numeric':
            try:
                val = Decimal(data)
            except:
                return False
            return True

    validate = classmethod(_validate)
```

Com poucas linhas, conseguimos aplicar esse modelo de objetos inicial no nosso programa.

Um detalhe importante é que o que foi feito nesta seção poderia ser feito de forma mais elegante com um recurso que não vimos, chamado *decorator* (ou decorador). Para não ter que explicar sobre decoradores neste ponto do livro, optamos por trabalhar com `classmethod()` e `staticmethod()` como funções.

6.9 Encapsulamento pythônico com a função property

Um tópico que ainda não cobrimos é encapsulamento *pythônico*. Para isso, vamos aprender a usar a função `property`.

Inicialmente, vamos imaginar um encapsulamento da leitura usando um *getter*. No caso da nossa classe `DataTable`, não queremos que o usuário acesse o nome pela referência de `_name`. O primeiro passo, portanto, é criar um *getter* e, depois, usar a função `property` para associá-lo à chamada de um atributo. Veja o exemplo para entender melhor:

```
class DataTable:
    def __init__(self, name):
        self._name = name
        self._columns = []
        self._references = []
        self._referenced = []
        self._data = []

    def _get_name(self):
        print("Getter executado!")
        return self._name

    name = property(_get_name)
```

Coloque essa modificação na classe `DataTable` e rode no console:

```
>>> table = DataTable("Empreendimento")
>>> table.name
```

```
Setter executado!
'Empreendimento'
```

O que aconteceu foi que, quando executamos `table.name`, o método `_get_name()` – que foi configurado como *getter* na função `property` – foi chamado. O mesmo princípio pode ser usado para o *setter* e *deleter*.

```
class DataTable:
    def __init__(self, name):
        self._name = name
        self._columns = []
        self._references = []
        self._referenced = []
        self._data = []

    def _get_name(self):
        print("Getter executado!")
        return self._name

    def _set_name(self, _name):
        print("Setter executado!")
        self._name = _name

    def _del_name(self):
        print("Deletter executado!")
        raise AttributeError("Não pode deletar esse atributo")

    name = property(_get_name, _set_name, _del_name)
```

Repare aqui que, na documentação, consta exatamente que a função `property` recebe nessa ordem: *getter*, *setter* e *deleter*. Agora salve em `domain.py` e execute no console Python:

```
>>> table = DataTable("Empreendimento")
>>> table.name
Getter executado!
'Empreendimento'
>>> table.name = 'Alerta'
Setter executado!
```

```
>>> del table.name
Deletter executado!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 16, in _del_name
AttributeError: Não pode deletar esse atributo
```

A atribuição de valor para `table.name` também foi feita por meio do método `_set_name()`, onde poderíamos ter colocado uma validação, por exemplo. O mesmo acontece para o *deleter*, que executa `_del_name()` por baixo.

Agora, podemos criar classes com encapsulamento *pythônico*.

6.10 Herança múltipla: herdando de várias classes

Nesta parte do livro, vou cobrir um assunto bem superficialmente, sem emprego prático, pois faz parte do contexto que estamos estudando de classes. Mais à frente, vamos usá-lo com um objetivo prático.

Python suporta uma forma de herança múltipla. Superficialmente, o que importa é a ordem de busca dos métodos. Vamos ver um exemplo e discuti-lo:
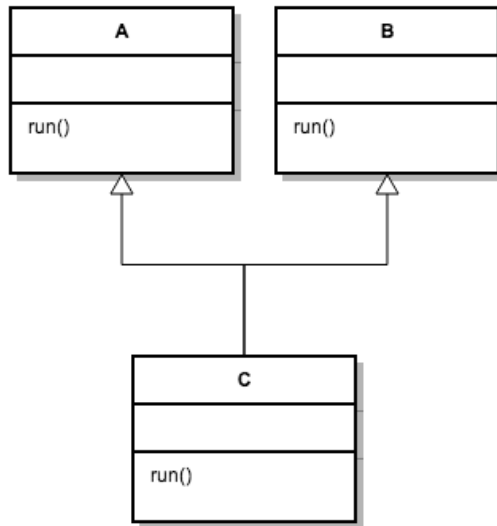
Fig. 6.1: Herança múltipla

```
class A:
    def run(self):
        return "A"

class B:
    def run(self):
        return "B"

class C(A, B):
    pass

a = A()
b = B()
c = C()
assert "A" == c.run()
```

O comportamento esperado é primeiro buscar `run()` na definição de `C`. Como não existe, o interpretador procura nas superclasses da esquerda para a direita, na definição da herança. Com isso, quando falamos `C(A, B)`, a

procura de `run()` é feita em `A` e em superclasses de `A`, e caso não exista, em `B` e superclasses de `B`.

No nosso caso, `c.run()` retorna `"A"`, já que o método foi encontrado em `A` antes de ser em `B`.

Essa ordem de busca de métodos é comumente chamada de *Method Resolution Order* (MRO). Podemos até acessá-la via um atributo especial:

```
>>> class A:
...     def run(self):
...         return "A"
>>> class B:
...     def run(self):
...         return "B"
>>> class C(A, B):
...     pass
>>> C.__mro__
(__main__.C, __main__.A, __main__.B, builtins.object)
```

O *output* nos diz a ordem: `C`, `A`, `B` e, no final, o próprio `builtins.object` de qual todas as classes herdam.

6.11 O que é DuckTyping?

Esse termo, *DuckTyping* é uma expressão que diz: tudo que pode fazer "quack" é um pato; por isso, *ducktyping* (em uma tradução livre, "a tipagem do pato"). Se colocarmos essa lógica em um contexto de objetos, o termo refere-se ao fato de que um objeto em Python pode se comportar como se fosse um outro objeto qualquer, desde que tenha determinada semelhança.

Em Python, quando passamos um objeto como parâmetro de uma função ou método, não há checagem de tipo, pois não existem tipos explícitos na assinatura, logo, qualquer objeto pode ser passado.

Se dentro do método acessamos o atributo `nome`, qualquer objeto que tenha esse atributo será compatível com esse código. Para métodos é a mesma coisa: se um método qualquer for chamado, qualquer objeto que tenha um método com mesmo nome e mesmo número de parâmetros poderá ser usado. Esse é o significado prático do *ducktyping*. Vamos exemplificar:

```
>>> from domain import *
>>> table = DataTable("Empreendimento")
>>>
>>> class DuckType:
...     pass
...
>>> duck = DuckType()
>>> duck.name = "quak"
>>>
>>> def print_name(table):
...     print(table.name)
...
>>> print_name(table)
Empreendimento
>>> print_name(duck)
quak
```

A função `print_name(tabela)` serve tanto para instâncias de `DataTable` quanto para qualquer outra que tenha o atributo `name`. Esse caso pode ser simples demais, mas em outros contextos, o *ducktyping* é muito empregado. Se por um lado é muito prático, por outro, seu uso indiscriminado pode levar até mesmo a bugs no código.

De qualquer forma, é uma característica relevante do universo de linguagens dinâmicas, inclusive Python.

6.12 Conclusão

Neste capítulo, exploramos os aspectos básicos de classes e objetos em Python. Descobrimos como declarar classes, que podemos adicionar atributos em tempo de execução, como são definidos os métodos de domínio, construtores e o famoso `__str__()`. Além disso, exploramos herança simples e múltipla, métodos estáticos e de classe, atributos de classe e encapsulamento com a função `property`.

A ideia não era explicar tudo, mas pelo menos uma parte inicial, permitindo, assim, que o nosso próximo passo seja criar código Python de qualidade para o mundo real.

Capítulo 7

Tratando erros e exceções: tornando o código mais robusto

7.1 Escreveu errado? Erros de sintaxe

Em Python, existem dois tipos principais de erros: os erros de sintaxe e as exceções.

Para utilizar os recursos de uma linguagem de programação, devemos aprender sua sintaxe. Se você tentou executar os exemplos do livro, talvez tenha visto um erro de sintaxe em algum momento. Esse tipo de erro impede a execução de um programa, ao passo que as exceções aparecem em grande maioria durante sua execução.

Veja o que acontece quando executamos um código com erro de sintaxe:

```
>>> print('Erro de sintaxe)
```

```
  File "<stdin>", line 1
    print('Erro de sintaxe)
                          ^
SyntaxError: EOL while scanning string literal
```

No exemplo anterior, ganhamos um `SyntaxError`, pois não fechamos corretamente a `string` que queremos imprimir na tela.

Agora, observe este outro exemplo:

```
>>> print(1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Aqui, vemos uma exceção sendo levantada em tempo de execução.

Qual o significado da exceção?

A exceção representa uma anomalia ou uma situação que precisa ser tratada especificamente.

Os erros de sintaxe, especificamente os que não podem ser *tratados*, devem ser corrigidos para que um programa possa ser executado. Um programa com esse tipo de erro não pode ser compilado para *bytecode* para ser executado.

Mesmo com a sintaxe correta, alguns erros podem acontecer com o programa rodando. Imagine, por exemplo, tentar ler um arquivo que não existe.

Em linguagens como C, muitas bibliotecas retornam -1 quando ocorrem erros, mas, em outras, existe uma forma especial de se tratá-los, as chamadas exceções. Python, assim como Java e Ruby, tem suporte para esse tratamento de exceções, geralmente introduzindo um comando. No caso de Python, é um comando composto (*compound statement*) que usa as palavras reservadas `try/except` para criar blocos de código que contêm alternativas para quando as exceções ocorrem.

7.2 Como tornar o código mais robusto?

Usando o exemplo anterior, como podemos contornar uma exceção gerada quando tentamos ler um arquivo que não existe? Nesse caso, vamos apenas

apresentar uma mensagem mais amigável. Veja o código a seguir:

```
import zipfile

banco_zip = zipfile.ZipFile("saida.zip")
banco_zip.extractall(path="banco")
banco_zip.close()
```

Aqui, contamos com a existência do arquivo `saida.zip`. Caso ele não exista, uma mensagem será exibida:

```
FileNotFoundError: [Errno 2] No such file or directory :
'saida.zip'
```

Esse é um erro facilmente tratável, usando o comando `try/except`. Na prática, o comando `try/except` é quem permite o tratamento de exceções em Python. Veja o exemplo:

```
import zipfile

try:
    banco_zip = zipfile.ZipFile("saida.zip")
    banco_zip.extractall(path="banco")
    banco_zip.close()
except FileNotFoundError:
    print("Arquivo inexistente")
```

Quando trabalhamos com arquivos, várias exceções podem ser levantadas. Na biblioteca padrão, temos uma hierarquia de exceções que são padronizadas e que nos permitem tratar os erros em níveis mais específicos ou genéricos. Vamos ver uma parte da árvore de exceções:

```
+-- OSError
|    +-- BlockingIOError
|    +-- ChildProcessError
|    +-- ConnectionError
|    |    +-- BrokenPipeError
|    |    +-- ConnectionAbortedError
|    |    +-- ConnectionRefusedError
|    |    +-- ConnectionResetError
```

```
|      +-- FileExistsError
|      +-- FileNotFoundError
|      +-- InterruptedError
|      +-- IsADirectoryError
|      +-- NotADirectoryError
|      +-- PermissionError
|      +-- ProcessLookupError
|      +-- TimeoutError
```

Vendo essa árvore, que representa uma hierarquia de exceções, conseguimos observar, por exemplo, que `OSError` é o que chamamos de *pai/mãe* de todas as outras exceções. Se quisermos tratar os erros de uma forma genérica, podemos usar `OSError` na parte do `except`.

Para tratar um erro mais especificamente, como um arquivo inexistente, usamos `FileNotFoundError` no bloco `except`. Repare que, se utilizarmos `OSError`, vamos tratar erros de todas as exceções *filhas*. Quando usamos `FileNotFoundError`, que não possui filhas, vamos tratar apenas este caso específico.

7.3 Tratando várias possíveis exceções em um mesmo bloco

Em muitos casos, o número de exceções que uma única linha de código pode gerar pode ser superior a 1. Por isso, é normal ter blocos que tratam diversas exceções simultaneamente.

Vamos ver um exemplo no qual tratamos, separadamente, mais de uma exceção:

```
import zipfile

try:
    banco_zip = zipfile.ZipFile("saida.zip")
    banco_zip.extractall(path="banco")
    banco_zip.close()
except FileNotFoundError:
    print("Arquivo inexistente")
```

```
except PermissionError as pme:
    print("Erro de permissao")
```

Se olharmos para a árvore de exceções, veremos que ambas são filhas de `OSError` e, por isso, poderíamos ter um tratamento único para ambas.

O que ficou faltando é que, apesar de tornar a mensagem de erro mais amigável, não estamos informando qual o arquivo que tentamos usar, o que gerou o erro.

Vamos ver uma solução que tem uma pequena variação na sintaxe, adicionando um *alias* para a instância da exceção usando a palavra reservada `as`:

```
import zipfile

try:
    banco_zip = zipfile.ZipFile("saidax.zip")
    banco_zip.extractall(path="banco")
    banco_zip.close()
except OSError as ose:
    print("Algum problema ao ler o arquivo {}"
          .format(ose.filename))
```

É muito importante entender que todas exceções que são filhas de `OSError` serão tratadas aqui. Todas elas possuem o atributo `filename`, portanto, esse código funcionará em todos os casos onde o erro é filho de `OSError`.

Em caso de dúvida, procure conhecer melhor o significado das exceções para tomar boas decisões quanto ao tratamento genérico, e não tratar casos demais sem querer. Não existe fórmula certa, logo, use o bom senso na hora de decidir em que nível de especificidade chegar.

Outra variação seria usar a sintaxe com tuplas:

```
import zipfile

try:
    banco_zip = zipfile.ZipFile("saida.zip")
    banco_zip.extractall(path="banco")
```

```
    banco_zip.close()
except (FileNotFoundError, PermissionError):
    print("Algum problema ao ler o arquivo")
```

7.4 Exceções e Python 3.3+

A partir do ramo 3.3, uma reorganização na hierarquia de exceções foi feita. Isso visou facilitar o tratamento de casos específicos e remover algumas inconsistências na linguagem.

A hierarquia completa está descrita na sequência.

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
```

```
     |    |    +-- ConnectionRefusedError
     |    |    +-- ConnectionResetError
     |    +-- FileExistsError
     |    +-- FileNotFoundError
     |    +-- InterruptedError
     |    +-- IsADirectoryError
     |    +-- NotADirectoryError
     |    +-- PermissionError
     |    +-- ProcessLookupError
     |    +-- TimeoutError
     +-- ReferenceError
     +-- RuntimeError
     |    +-- NotImplementedError
     +-- SyntaxError
     |    +-- IndentationError
     |         +-- TabError
     +-- SystemError
     +-- TypeError
     +-- ValueError
     |    +-- UnicodeError
     |         +-- UnicodeDecodeError
     |         +-- UnicodeEncodeError
     |         +-- UnicodeTranslateError
     +-- Warning
          +-- DeprecationWarning
          +-- PendingDeprecationWarning
          +-- RuntimeWarning
          +-- SyntaxWarning
          +-- UserWarning
          +-- FutureWarning
          +-- ImportWarning
          +-- UnicodeWarning
          +-- BytesWarning
          +-- ResourceWarning
```

Em Python, toda exceção deve derivar em algum nível de `BaseException`. As exceções definidas pelo usuário, as chamadas *user-defined exceptions* geralmente derivam de `Exception` ou subclasses como a própria `OSError`. Especialmente a partir do Python 3.3, parte da

hierarquia foi refatorada e diversos erros de mais alto nível foram criados, permitindo códigos mais claros quanto ao tratamento de erro implementado.

7.5 Executando código se nenhuma exceção for levantada

Se pegarmos os exemplos anteriores, podemos observar que, se uma das duas primeiras linhas levantar um erro, não conseguiremos saber exatamente onde foi que ele aconteceu, a não ser que olhemos todo o *traceback*. Podemos dizer que o método `extractall` só vai ser executado caso a instanciação da classe `ZipFile` não dê nenhum erro. Dessa forma, poderemos expressar:

```
import zipfile

try:
    banco_zip = zipfile.ZipFile("saida.zip")
except (FileNotFoundError, PermissionError):
    print("Algum problema ao ler o arquivo")
else:
    banco_zip.extractall(path="banco")
```

O bloco `else` garantiu que `extractall` seja executado somente caso a operação anterior – de abertura do arquivo – não levante uma exceção. Aqui, expressamos que o método `extractall` depende de um arquivo válido aberto, que vem da chamada ao construtor `ZipFile`.

É claro que, dentro do bloco `else`, a chamada do método também pode gerar um erro. O importante é sempre ter consciência ao decidirmos sobre a implementação do tratamento de erros. Muitas vezes o emprego do `else` pode tornar o código mais claro.

7.6 Ações de limpeza

Quando trabalhamos com códigos que podem gerar exceções, muitas vezes estamos manipulando *recursos* abertos anteriormente (conexão com banco de dados, por exemplo), e queremos garantir que eles sejam finalizados (ou fechados) independente da ocorrência, e não de alguma exceção. Para isso,

temos a palavra reservada `finally` para definir um bloco que sempre será executado, permitindo que esse tipo de código chame os métodos de finalização e liberação corretamente.

As *ações de limpeza* (*clean-up actions*) serão executadas **sempre**, independente de terem ocorrido erros ou não. Seguindo a lógica do nosso exemplo, poderíamos ter um código semelhante ao a seguir:

```
import zipfile

banco_zip = None

try:
    banco_zip = zipfile.ZipFile("saida.zip")
    banco_zip.extractall(path="banco")
except PermissionError:
    print("Algum problema ao extrair o arquivo")
finally:
    banco_zip.close()
```

Assim, garantimos que `banco_zip.close()` sempre seja executado, não deixando arquivos abertos no nosso programa.

7.7 Comando raise: levantando exceções

No domínio da nossa aplicação, existem 3 tipos de dados, cada um representado nos arquivos de metadados, como: `'bigint'`, `'varchar'` e `'numeric'`. Qualquer valor é considerado inválido, além desses três.

Essa restrição de negócio pode ser representada por meio do lançamento de uma exceção, justamente quando o valor for inválido. Esse método será privado na nossa API, mas será utilizado por diversos outros métodos que recebem o tipo do dado e precisam validar se está correto. Quando queremos levantar uma exceção, usamos o comando `raise` e, em seguida, passamos uma instância ou uma classe de uma exceção. Veja o exemplo:

```
def validate_kind(kind):
    if not kind in ('bigint', 'numeric', 'varchar'):
        raise Exception("Tipo inválido")
```

Sempre que uma informação de tipo for lida dos arquivos de metadados, podemos executar essa função no valor e verificar se ele é valido, diante das restrições do código. No nosso exemplo, usamos a própria `Exception` da biblioteca padrão, mas poderíamos ter usado uma *user-defined* `exception`. O mais importante é que a exceção levantada seja coerente com a situação.

Veja uma variação do código referenciando uma classe de exceção definida por usuário:

```
class InvalidDataTypeException(Exception):
    pass

def validate_kind(kind):
    if not kind in ('bigint', 'numeric', 'varchar'):
        raise InvalidDataTypeException
```

Levantando uma mesma exceção tratada

Em alguns casos, o nosso código quer apenas ter conhecimento de que uma exceção foi levantada, porém, quer continuar propagando-a para o código cliente. Outro exemplo são filtros de aplicações web, logando exceções, mas não escondendo do resto do código.

Veja o código a seguir:

```
class InvalidDataTypeException(Exception):
    pass


def validate_kind(kind):
    if not kind in ('bigint', 'numeric', 'varchar'):
        raise InvalidDataTypeException

try:
    validate_kind('invalid type')
except Exception as e:
    print("Peguei o erro mas vou repassa-lo")
    raise e
```

Nesse caso, o código que chama o trecho anterior continua recebendo a exceção levantada pela função `validate_kind`. Esse tipo de código é muito

comum quando queremos apenas saber que um determinado erro aconteceu, mas não queremos afetar o fluxo do programa com um tratamento mais específico.

7.8 Exemplo de um tratamento de erros mais robusto em Python

Conhecendo melhor o mecanismo de tratamento de exceções em Python, poderíamos ter escrito a função `main()` em nosso primeiro programa do capítulo 4, da seguinte forma:

```
# coding: utf-8

import io
import sys
import urllib.request as request

BUFF_SIZE = 1024


def download_length(response, output, length):
    times = length // BUFF_SIZE
    if length % BUFF_SIZE > 0:
      times += 1
    for time in range(times):
      output.write(response.read(BUFF_SIZE))
      print("Downloaded %d" % (((time * BUFF_SIZE)/length)*100))


def download(response, output):
    total_downloaded = 0
    while True:
      data = response.read(BUFF_SIZE)
      total_downloaded += len(data)
      if not data:
          break
      out_file.write(data)
      print('Downloaded {bytes}'.format(bytes=total_downloaded))
```

```
def main():
    response = request.urlopen(sys.argv[1])
    out_file = io.FileIO("saida.zip", mode="w")
    content_length = response.getheader('Content-Length')

    try:
      if content_length:
          length = int(content_length)
          download_length(response, out_file, length)
      else:
          download(response, out_file)
    except Exception as e:
      print("Erro durante o download do arquivo {}"
            .format(sys.argv[1]))
    finally:
      out_file.close()
      response.close()

    print("Fim")
```

A grande diferença dessa versão para a versão sem tratamento de erros é que uma mensagem amigável é escrita na saída padrão, caso ocorra alguma exceção na função de download executada e nenhum recurso seja mantido aberto, mesmo em situação de exceção. Com a cláusula `finally`, garantimos que os dois recursos abertos sejam fechados, independente da ocorrência de erros ou não.

7.9 Conclusão

O mecanismo de exceções em Python não é tão diferente de outras linguagens, e a sintaxe também é bem familiar, de certa forma. Basicamente, temos o comando/bloco `try/except` com cláusulas opcionais `else` e `finally`. A existência de uma hierarquia também é comum em outras linguagens e não é muito diferente em Python. Também não existe nenhuma exigência em tempo de compilação ou execução quando tratamos determinadas exceções. Vimos como levantar exceções e até mesmo como repassá-las a outro

código.

Agora, já podemos criar tratamentos de erro elaborados e, por consequência, programas mais robustos. É importante saber empregar o que existe corretamente. Ao longo do livro, outros exemplos mostrarão tratamento de exceções, e explicaremos a motivação para a implementação.

No próximo capítulo, vamos ver *classes*. Python tem um bom suporte para o trabalho com classes e objetos, o paradigma Orientação a Objetos. Porém, por características da linguagem, acabamos tendo muitos detalhes que precisam ser abordados com calma para melhor compreensão.

Capítulo 8

Testes em Python: uma prática saudável

Em minha opinião, aprender a criar os testes logo cedo contribui para que o desenvolvedor crie a cultura de escrever código com testes, de preferência, sempre.

Python possui um ecossistema muito rico de ferramentas de teste. Para começar, o módulo de testes `unittest` já está na biblioteca padrão, inclusive com um módulo de *mocking* em `unittest.mock`. Ou seja, se você tem o interpretador instalado, já tem tudo necessário para escrever códigos de teste.

Do pouco código que acumulamos até agora, temos alguns testes unitários bem simples a serem feitos e outros que vão exigir mocks e outros recursos. Vamos analisar caso a caso.

A ideia deste capítulo é passar o básico do pacote `unittest`, que é distri-

buído na biblioteca padrão. Por ter um ecossistema de testes rico, ele permite testes até mesmo sem o uso desse pacote. Outra prática comum é usar *Test Runners* para poupar o trabalho de configurar programaticamente as suítes e testes que rodarão.

Neste capítulo, vamos ver como criar código de testes, como rodá-los e alguns outros detalhes relacionados.

8.1 Criando o primeiro TestCase

O módulo `unittest` tem o design inspirado no `Junit`, ferramenta tradicional de testes em Java. O primeiro passo para fazermos um teste da forma mais tradicional é criar uma classe que herda de `unittest.TestCase`.

Vamos criar um caso de testes para a classe `Column` do nosso aplicativo. Essa classe possui um método construtor e um de domínio, chamado `validate()`. Faremos o teste na sequência.

Relembrando a classe Column

Um aspecto muito importante no nosso aplicativo é a *consistência*. Queremos que os dados lidos estejam em conformidade com as suas descrições. Se um atributo tem tipo `bigint`, então deve aceitar apenas valores inteiros. Esse aspecto nos motiva a criar o nosso primeiro teste.

O nosso primeiro alvo será a classe `Column`, que, além de guardar os atributos, é responsável por validar os tipos suportados pelo nosso programa. Os parâmetros são o nome do tipo e um valor. Se o valor for compatível com o tipo, a função retorna `True`. Por exemplo: o inteiro `1000` é compatível com o tipo `'bigint'`, mas não com `'varchar'`.

Veja o código:

```
import decimal
import unittest

class Column:
    def __init__(self, name, kind, description=""):
        self._name = name
        self._kind = kind
```

```
        self._description = description

    @staticmethod
    def validate(kind, data):
        if kind == 'bigint':
            if isinstance(data, int):
                return True
            return False
        elif kind == 'varchar':
            if isinstance(data, str):
                return True
            return False
        elif kind == 'numeric':
            try:
                val = decimal.Decimal(data)
            except:
                return False
            return True
```

Agora, temos que criar testes para garantir que o comportamento esperado aconteça. Aqui, separei o teste da função `validate()` em três partes, mas também poderíamos ter apenas uma função de teste. Veja o código:

```
import decimal
import unittest

class Column:
    def __init__(self, name, kind, description=""):
        self._name = name
        self._kind = kind
        self._description = description

    @staticmethod
    def validate(kind, data):
        if kind == 'bigint':
            if isinstance(data, int):
                return True
            return False
        elif kind == 'varchar':
```

```
            if isinstance(data, str):
                return True
            return False
        elif kind == 'numeric':
            try:
                val = decimal.Decimal(data)
            except:
                return False
            return True

class ColumnTest(unittest.TestCase):
    def test_validate_bigint(self):
        self.assertTrue(Column.validate('bigint', 100))
        self.assertTrue(not Column.validate('bigint', 10.1))
        self.assertTrue(not Column.validate('bigint', 'Texto'))

    def test_validate_numeric(self):
        self.assertTrue(Column.validate('numeric', 10.1))
        self.assertTrue(Column.validate('numeric', 100))
        self.assertTrue(not Column.validate('numeric', 'Texto'))

    def test_validate_varchar(self):
        self.assertTrue(Column.validate('varchar', 'Texto'))
        self.assertTrue(not Column.validate('varchar', 100))
        self.assertTrue(not Column.validate('varchar', 10.1))

if __name__ == "__main__":
    unittest.main()
```

Primeiro, criamos uma classe de teste chamada **caso de teste** (*testcase*). Essa classe deve herdar obrigatoriamente de `unittest.TestCase`. Depois, chamamos a função `main()` do módulo `unittest`.

A seguir, veremos como invocar o mecanismo de testes.

8.2 Rodando um arquivo com testes

Por enquanto, por conveniência, vamos colocar no mesmo arquivo o código da classe `Column` e o teste `ColumnTest`. Salve o arquivo como

`test_column.py`. Rode-o:

```
$ python test_column.py
...
----------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

Todos os testes rodam com sucesso. Vamos entender tudo o que aconteceu para rodá-los, analisando os trechos de código separadamente.

Teste de chamanda na linha de comando

O código a seguir, que fica no final do arquivo `test_column.py`, é verdadeiro, ou seja, `__name__` é igual à string `"__main__"`, portanto a chamada `unittest.main()` é feita.

A função `main()` do módulo `unittest` procura no próprio módulo `__main__` – o que foi chamado – por classes que herdam de `TestCase` e executa os testes. Outras classes de testes importadas por esse arquivo também seriam executadas.

Veja o código que faz o que foi descrito logo a seguir:

```
>>> if __name__ == "__main__":
...     unittest.main()
```

Código do teste

A outra parte relevante é a definição do *testcase* que é encontrado e executado.

```
# coding: utf-8

from domain import Column

class ColumnTest(unittest.TestCase):
    def test_validate_bigint(self):
        self.assertTrue(Column.validate('bigint', 100))
        self.assertTrue(not Column.validate('bigint', 10.1))
```

```
        self.assertTrue(not Column.validate('bigint', 'Texto'))

    def test_validate_numeric(self):
        self.assertTrue(Column.validate('numeric', 10.1))
        self.assertTrue(Column.validate('numeric', 100))
        self.assertTrue(not Column.validate('numeric', 'Texto'))

    def test_validate_varchar(self):
        self.assertTrue(Column.validate('varchar', 'Texto'))
        self.assertTrue(not Column.validate('varchar', 100))
        self.assertTrue(not Column.validate('varchar', 10.1))
```

Aqui, dois detalhes são importantes: apenas métodos com prefixo `test_` serão executado pelo mecanismo de testes, e eles têm de estar em classes que herdam de `TestCase`. Cada método encontrado que atende esses requisitos é executado como um teste. No nosso *testcase*, temos 3 métodos que os atendem e, por isso, na saída é informado que 3 testes rodaram com sucesso.

A API de testes fica na própria instância – `self` – do caso de teste e é ela que possui os métodos de asserção. **Asserção** é o nome dado à verificação de valores, que gera um erro nos teste caso seja falsa.

A seguir, você pode ver alguns dos métodos que estão disponíveis. Os mais básicos são:

```
self.assertEqual(a, b)           # a == b
self.assertNotEqual(a, b)        # a != b
self.assertTrue(x)               # bool(x) is True
self.assertFalse(x)              # bool(x) is False
self.assertIs(a, b)              # a is b
self.assertIsNot(a, b)           # a is not b
self.assertIsNone(x)             # x is None
self.assertIsNotNone(x)          # x is not None
self.assertIn(a, b)              # a in b
self.assertNotIn(a, b)           # a not in b
self.assertIsInstance(a, b)      # isinstance(a, b)
self.assertNotIsInstance(a, b)   # not isinstance(a, b)
```

8.3 API de linha de comando para testes

No exemplo anterior, consideramos que a chamada `unittest.main()` inicia o processo de rodar testes. Porém, existe uma limitação muito ruim nessa forma de iniciação: para os testes rodarem, temos que passar o nome do arquivo na linha de comando para que eles sejam executados.

E se quiséssemos rodar testes de vários módulos?

Felizmente, temos uma API de linha de comando que permite que os testes sejam descobertos e executados. Essa API ainda possui uma série de parâmetros que atendem a várias necessidades.

Para simular a `unittest.main()`, você precisa apenas informar o nome do módulo. Lembre-se de que qualquer `TestCase` declarado ou importado no módulo especificado rodará. No nosso caso, temos apenas um caso de teste com três métodos.

```
$ python -m unittest test_column -v
test_validate_bigint (test_column.ColumnTest) ... ok
test_validate_numeric (test_column.ColumnTest) ... ok
test_validate_varchar (test_column.ColumnTest) ... ok

----------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Você pode especificar um caso de teste, como `test_column.ColumnTest`, ou um método de um caso de teste, por exemplo `test_column.ColumnTest.test_validate_bigint`. Veja o exemplo do segundo caso:

```
$ python -m unittest
                test_column.ColumnTest.test_validate_bigint -v
test_validate_bigint (test_column.ColumnTest) ... ok

----------------------------------------------------------------
Ran 1 tests in 0.000s

OK
```

É possível especificar vários módulos de uma vez, apenas listando-os separados por espaço:

```
$ python -m unittest test_column test_data_table -v
test_add_relationship (test_data_table.DataTableTest) ... ok
test_add_reverse_relationship (test_data_table.DataTableTest)
... ok
test_validate_bigint (test_column.ColumnTest) ... ok
test_validate_numeric (test_column.ColumnTest) ... ok
test_validate_varchar (test_column.ColumnTest) ... ok

------------------------------------------------------------------
Ran 5 tests in 0.001s

OK
```

É possível rodar o comando `python -m unittest` sem mais parâmetros, e todos os testes rodarem. Porém, na configuração padrão, existe a restrição de que os arquivos de testes tenham o prefixo `test_`, como `test_column.py`. Você pode alterar esse padrão passando um parâmetro para o subcomando `discover`. Veja o exemplo que funcionaria com o arquivo se ele fosse renomeado para `column_test.py`:

```
$ python -m unittest discover -p '*_test.py'
...
------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Diante dessas opções, talvez um bom ponto de partida seja adotar a convenção padrão: usar o prefixo `test_` e chamar o comando sem parâmetros, como `python -m unittest`. Desta forma, o próprio `unittest` procurará os módulos de teste e os casos de testes definidos, e rodará todos eles, sem exigir nenhum código, além das classes de caso de testes.

Apesar de muito flexível, controlar as configurações pela linha de comando nem sempre é o que queremos. Em alguns casos, queremos usar as

configurações programáticas para modificar a execução de etapas desse processo de execução dos testes. Um exemplo clássico é alterar o formato da saída.

8.4 API da biblioteca unittest

Algumas customizações no mecanismo de testes não podem ser feitas pela linha de comando. Por isso, o `unittest` também possui toda uma API programática para configuração e execução do mecanismo de testes.

Pela linha de comando não fica tão claro, mas temos duas etapas bem definidas nesse processo: descoberta/carregamento dos testes e execução.

A descoberta/carregamento (`discovery`/ `loading`) é a etapa na qual os testes são descobertos e carregados pelo `TestLoader`. O principal objetivo dessa classe é ajudar com a criação das suítes de testes `unittest.suite.TestSuite`. Podemos criar uma suíte carregando direto de um módulo de testes. Veja o exemplo:

```
import unittest
import test_column

suite = unittest.TestLoader().loadTestsFromModule(test_column)
unittest.TextTestRunner(verbosity=2).run(suite)
```

O método `discover(start_dir, pattern='test*.py', top_level_dir=None)` procura os testes em módulos, buscando-os em um diretório inicial e, recursivamente, nas pastas dentro dele. Ele também retorna uma suíte de testes.

```
import unittest

suite = unittest.TestLoader().discover('.')
unittest.TextTestRunner(verbosity=2).run(suite)
```

Na etapa de execução dos testes, o executor (ou `Runner`) precisa de um objeto do tipo suíte de testes para rodar. No nosso exemplo, escolhemos a implementação `TextTestRunner` para rodar a nossa suíte.

```
>>> import unittest
>>> suite = unittest.TestLoader()
                    .loadTestsFromModule(test_column)
>>> unittest.TextTestRunner(verbosity=2).run(suite)
test_validate_bigint (test_column.ColumnTest) ... ok
test_validate_numeric (test_column.ColumnTest) ... ok
test_validate_varchar (test_column.ColumnTest) ... ok

-----------------------------------------------------------------
Ran 3 tests in 0.001s

OK
<unittest.runner.TextTestResult run=5 errors=0 failures=0>
```

Tudo o que é feito pela linha de comando pode ser feito por meio da API, manipulando as classes de `loaders` e `runners`.

A grande diferença é que agora podemos substituir as implementações da biblioteca padrão por implementações customizadas, e controlar ainda mais o mecanismo.

8.5 Customizando saída para HTML

Como já sabemos configurar o mecanismo de testes para rodar de acordo com nossa necessidade, vamos customizá-lo um pouco mais para exemplificar como podemos empregar essa facilidade.

Em minha opinião, a facilidade de customização do mecanismo de testes é um ponto muito forte da linguagem. Em alguns casos, podemos querer exportar o resultado da suíte de testes para algum outro programa. A saída padrão, mesmo tendo uma boa padronização, não é flexível como os formatos `JSON`, ou até mesmo `HTML`.

Em vez de gerar um `output` texto, geraremos um código HTML que poderá ser aberto no navegador. Para isso, precisamos implementar nossos próprios `runners` e `TestResults`.

Como vimos, o `runner` é o responsável por exibir o resultado dos testes para o usuário. O que fizemos foi implementar um `runner` que imprime um `html` com o resultado dos testes em vez da saída padrão. Também precisamos

de uma outra classe auxiliar HTMLResult que nos permite exibir, na saída, mais informações sobre os testes que não falharam ou geraram erro.

Vamos novamente analisar cada parte separadamente na sequência.

8.6 Implementando um TestRunner customizado

A função do Runner é configurar um TestResult e executar os testes, passando o TestResult como parâmetro. A execução dos testes é feita por meio da chamada run() no objeto test, que é uma instância de TestSuite dentro do método também chamado run(), só que agora, do HTMLTestRunner. Essa execução invocará diversos métodos no TestResult para notificar sobre os testes com sucesso, erros e falhas.

Para gerar o HTML, iteramos na lista result.infos

```
import unittest

HTML = '''\
<html>
  <head>
    <title>unittest output</title>
  </head>
  <body>
    <table>
{}
    </table>
  </body>
</html>'''

OK_TD = '<tr><td style="color: green;">{}</td></tr>'
ERR_TD = '<tr><td style="color: red;">{}</td></tr>'

class HTMLTestResult(unittest.TestResult):
    def __init__(self, runner):
        unittest.TestResult.__init__(self)
        self.runner = runner
        self.infos = []
```

```
        self.current = {}

    def newTest(self):
        self.infos.append(self.current)
        self.current = {}

    def startTest(self, test):
        self.current['id'] = test.id()

    def addSuccess(self, test):
        self.current['result'] = 'ok'
        self.newTest()

    def addError(self, test, err):
        self.current['result'] = 'error'
        self.newTest()

    def addFailure(self, test, err):
        self.current['result'] = 'fail'
        self.newTest()

    def addSkip(self, test, err):
        self.current['result'] = 'skipped'
        self.current['reason'] = err
        self.newTest()

class HTMLTestRunner:
    def run(self, test):
        result = HTMLTestResult(self)
        test.run(result)
        table = ''
        for item in result.infos:
            if item['result'] == 'ok':
                table += OK_TD.format(item['id'])
            else:
                table += ERR_TD.format(item['id'])

        print(HTML.format(table))
        return result
```

```
if __name__ == "__main__":
    suite = unittest.TestLoader().discover('.')
    HTMLTestRunner().run(suite)
```

O nosso `TestResult`, chamado de `HTMLTestResult`, implementa alguns métodos que serão chamados nesse objeto, de forma que ele seja o responsável em guardar os resultados dos testes. Esses métodos compõem um protocolo de notificação dos resultados. Para o nosso exemplo, escolhemos os métodos `startTest(test)`, `addSuccess(test)`, `addError(test)`, `addFailure(test)` e `addSkip(test)`. Além disso, usamos algumas variáveis de instância e um método auxiliar para implementar essa tarefa.

O `newTest` é um método auxiliar que foi criado para pegar o último teste modificado e colocar em uma lista, que será usada pelo nosso `runner`. O método `startTest(test)` é executado um pouco antes de o teste – contido no parâmetro `test` – ser executado. Nesse momento, pegamos o identificador do teste e guardamos no dicionário `current`.

Os próximos 4 métodos são chamados para notificar o resultado da execução do teste. Cada função está associada a um evento, no caso: sucesso, erro, falha ou pulo do teste. Eles também colocam o resultado no dicionário `current` e chamam `newTest` para que esse teste seja adicionado na lista de resultados e o espaço para o próximo teste seja aberto.

```
class HTMLTestResult(unittest.TestResult):
    def __init__(self, runner):
        unittest.TestResult.__init__(self)
        self.runner = runner
        self.infos = []
        self.current = {}

    def newTest(self):
        self.infos.append(self.current)
        self.current = {}

    def startTest(self, test):
        self.current['id'] = test.id()
```

```
    def addSuccess(self, test):
        self.current['result'] = 'ok'
        self.newTest()

    def addError(self, test, err):
        self.current['result'] = 'error'
        self.newTest()

    def addFailure(self, test, err):
        self.current['result'] = 'fail'
        self.newTest()

    def addSkip(self, test, err):
        self.current['result'] = 'skipped'
        self.current['reason'] = err
        self.newTest()

if __name__ == "__main__":
    suite = unittest.TestLoader().discover('.')
    HTMLTestRunner().run(suite)
```

Agora, usamos o método `discover()` da suíte para descobrir os testes no diretório corrente, e a executamos com nosso `HTMLTestRunner`. A saída é um código HTML, em texto, com o resultado dos testes.

```
<html>
  <head>
    <title>unittest output</title>
  </head>
  <body>
    <table>
      <tr>
        <td style="color: green;">
          test_domain.ColumnTest.test_validate_as_static_method
        </td>
      </tr>
    </table>
  </body>
</html>
```

Uma saída em html renderizado:

test_column.ColumnTest.test_validate_bigint
test_column.ColumnTest.test_validate_numeric
test_column.ColumnTest.test_validate_varchar

Fig. 8.1: Saída html

Esse exemplo foi feito para demonstrar um pouco mais da flexibilidade do mecanismo de testes e para explorar um pouco mais a API programática do pacote `unittest`. A seguir, vamos focar em como criar o código que, de fato, vai fazer os testes no seu programa ou biblioteca.

8.7 Testando erros: quando o código joga exceções

Muitas vezes, temos um método ou função que pode jogar uma exceção em determinados casos. Uma função retorna um resultado caso os parâmetros estejam corretos, e um erro caso os parâmetros não sejam válidos. O ideal é que ambos os casos sejam testados.

Se voltarmos à nossa aplicação, na classe `DataTable` podemos ver que o método `add_column(name, kind)` pode jogar uma exceção caso `kind` seja inválido. Vamos refatorar esse método para que ele jogue a exceção caso `kind` não seja um tipo conhecido. Depois vamos criar o teste para o método `add_column()`. No código a seguir, omitimos os métodos que não são usados no teste.

```
from domain import Column

class DataTable:
    def add_column(self, name, kind, description=""):
        self._validate_kind(kind)
        column = Column(name, kind, description=description)
        self._columns.append(column)
        return column
```

```
    def _validate_kind(self, kind):
        if not kind in ('bigint', 'numeric', 'varchar'):
            raise Exception("Tipo inválido")
```

Agora, temos dois casos para testar: um quando `kind` é um dos tipos suportados, e outro quando não for. Vamos ver os testes:

```
from domain import Column

class DataTable:
    def add_column(self, name, kind, description=""):
        self._validate_kind(kind)
        column = Column(name, kind, description=description)
        self._columns.append(column)
        return column

    def _validate_kind(self, kind):
        if not kind in ('bigint', 'numeric', 'varchar'):
            raise Exception("Tipo inválido")

class DataTableTest(unittest.TestCase):
    def test_add_column(self):
        self.assertEqual(0, len(self.table._columns))

        self.table.add_column('BId', 'bigint')
        self.assertEqual(1, len(self.table._columns))

        self.table.add_column('value', 'numeric')
        self.assertEqual(2, len(self.table._columns))

        self.table.add_column('desc', 'varchar')
        self.assertEqual(3, len(self.table._columns))

    def test_add_column_invalid_type(self):
        a_table = DataTable('A')
        self.assertRaises(Exception,
            a_table.add_column, ('col', 'invalid'))
```

O `assertRaises` precisa receber o método que será

testado e seus argumentos. Aqui, internamente, dentro de `assertRaises()`, temos a aplicação prática do conceito de *packing*, pois o terceiro argumento de `assertRaises` será usado na chamada do método `table.add_column`. Vamos ter algo como `table.add_column(*('col', 'invalid'))` sendo executado, que, no fundo, é o mesmo que `table.add_relationship('col', 'invalid')`.

Outra forma de testar que esse erro é levantado é usando um bloco `try/except` combinado com a função `fail()` do `TestCase`. Veja o exemplo e depois a explicação, a seguir:

```
class DataTableTest(unittest.TestCase):
    def test_add_column_invalid_type_fail(self):
        a_table = DataTable('A')
        error = False

        try:
            a_table.add_column('col', 'invalid')
        except:
            error = True

        if not error:
            self.fail("Chamada não gerou erro, mas deveria")
```

Como o método `add_column` levanta um erro, pois está com os parâmetros inválidos, a variável `error` passa a ser `True`, e a linha `self.fail(msg)` **não** é executada. Se o parâmetro for válido, o método não levanta erro, e a variável `error` fica com valor `False`, executando, assim, a chamada ao método `fail(msg)`. Este é o objetivo desse teste: queremos ter certeza de que a chamada gerou um erro.

A chamada faz com que o teste falhe e exiba a mensagem passada como parâmetro. É uma outra forma de testar situações em que erros deveriam ser levantados.

Ainda existe outra maneira, que veremos mais à frente, pois introduz um comando ainda não trabalhado.

8.8 Inicialização e finalização: setUp e tearDown

Um princípio importante em testes é o de isolamento. Deve existir um isolamento entre os ambientes de teste, em cada método de teste. Ou seja, as mudanças de ambiente – como criação de variáveis, alterações em objetos importados e outras – não devem ser vistas por outro método de teste que não seja o qual as realizou.

Por outro lado, é comum que muitos testes precisem de uma variável do objeto que possivelmente está sendo testado, que sempre é iniciada em um estado inicial igual. Pelo princípio do isolamento, se um teste modifica o valor dessa variável, ela não poderia estar modificada no início do próximo teste.

Veja o exemplo de dois testes que usam uma inicialização idêntica de objeto:

```
class DataTableTest(unittest.TestCase):
    def test_add_column(self):
        table = DataTable('A')
        assertEqual(0, len(self.table._columns))

        table.add_column('BId', 'bigint')
        assertEqual(1, len(self.table._columns))

        table.add_column('value', 'numeric')
        assertEqual(2, len(self.table._columns))

        table.add_column('desc', 'varchar')
        assertEqual(3, len(self.table._columns))

    def test_add_column_invalid_type(self):
        table = DataTable('A')
        assertRaises(Exception,
            self.table.add_column, ('col', 'invalid'))
```

Repare que, no exemplo anterior, primeiramente todos os testes precisam de um `DataTable` no mesmo estado inicial. Aqui, respeitamos o isolamento, já que o ambiente de cada teste tem sua variável criada no seu início, mas se

tivéssemos mais métodos de testes, teríamos mais linhas iguais. Vamos ver como podemos melhorar isso a seguir.

Os métodos `setUp` e `tearDown` são executados, respectivamente, antes e depois de cada teste rodar. Com eles, podemos reaproveitar código de inicialização e finalização que serão rodados para cada método de teste. A grande vantagem que eles nos dão é que podemos criar um código único de inicialização e finalização para cada método de teste. Isso nos permite tanto ter o isolamento quanto a eliminação de código duplicado. Veja a seguir como podemos melhorar o exemplo anterior:

```
class DataTableTest(unittest.TestCase):
    def setUp(self):
        self.table = DataTable('A')

    def test_add_column(self):
        self.assertEqual(0, len(self.table._columns))

        self.table.add_column('BId', 'bigint')
        self.assertEqual(1, len(self.table._columns))

        self.table.add_column('value', 'numeric')
        self.assertEqual(2, len(self.table._columns))

        self.table.add_column('desc', 'varchar')
        self.assertEqual(3, len(self.table._columns))

    def test_add_column_invalid_type(self):
        self.assertRaises(Exception,
            self.table.add_column, ('col', 'invalid'))
```

Em vez de criar uma instância dentro de todos os métodos, usamos uma criada no método `setUp`, que sempre está no mesmo estado antes da execução de cada método. Foi exatamente o que fizemos no exemplo anterior.

O método `tearDown()` é executado após cada método de teste. Uma ação muito comum é fechar arquivos ou conexões com banco de dados, mas várias outras tarefas podem ser executadas nele. No final das contas, para cada método de teste executado, temos uma sequência como: `setup()`,

test_minha_funcao(), tearDown(). É importante que esse protocolo seja usado sempre quando for vantajoso, tanto para o isolamento quanto para a diminuição de código repetido.

Exceções no método setUp()

Um detalhe importante é que, se por acaso o método setUp() levantar algum erro não tratado, o método tearDown() **não** será executado. Isso pode nos trazer problemas!

Imagine que você tem um teste (de integração) que abre conexão com duas fontes de dados. Poderíamos colocar no setUp() a criação dos objetos que nos dão acesso a esses recursos. O problema é que abrimos o primeiro recurso com sucesso, mas, se na hora de abrirmos o segundo, o código levantar uma exceção, ele não será executado caso a finalização dos recursos esteja no tearDown().

Você poderia resolver isso com um bloco try/except, porém, existe uma forma mais adequada de tratar essas situações. Felizmente, há uma maneira de rodar alguma função, mesmo que o método setUp levante erros não tratados.

8.9 Ações de limpeza nos testes unitários: cleanup actions

Nesta parte do livro, o foco está mais em mostrar o funcionamento do que discutir a aplicação. Uma motivação foi dada anteriormente, na qual temos a hipótese de que um recurso que deveria ser finalizado no tearDown() acaba não o sendo, pois pode ter ocorrido um erro não tratado no setUp().

Para que isso não seja um problema, podemos configurar ações de limpeza (*cleanup actions*), que serão executadas mesmo que um erro tenha sido levantado no setUp() ou no tearDown().

Veja o exemplo a seguir:

```
import unittest

class DataTableTest(unittest.TestCase):
```

```
    def setUp(self):
        self.out_file = open()

    def test_add_column(self):
        pass

    def tearDown(self):
        print("tearDown")
```

Esse exemplo nunca executará o método `tearDown()`, porque a chamada à função `open()` levanta um erro. Nessa situação, `tearDown()` não roda. Assim, precisamos adicionar uma ação de limpeza. Essas ações serão executadas sempre depois do `tearDown()`, mesmo que ele não rode. Veja o exemplo atualizado:

```
import unittest

class DataTableTest(unittest.TestCase):
    def setUp(self):
        self.addCleanup(self.my_cleanup, ('cleanup executado'))
        self.out_file = open()

    def my_cleanup(self, msg):
        print(msg)

    def test_add_column(self):
        pass

    def tearDown(self):
        print("Nunca executado")
```

Rode na linha de comando e veja o resultado:

```
$ python -m unittest -v 08_10_cleanup.py
cleanup executado
E
=================================================================
ERROR: test_add_column (08_10_cleanup.DataTableTest)
-----------------------------------------------------------------
```

```
Traceback (most recent call last):
  File
   "/Users/felipecruz/Projects/livros/exemplos/08_10_cleanup.py",
   line 6, in setUp
    self.out_file = open()
TypeError: Required argument 'file' (pos 1) not found

-------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Vemos que a string `cleanup executado` é impressa na tela. Isso porque o método `my_cleanup()` foi executado, já que foi registrado como uma *cleanup action* na chamada `self.addCleanup()`, passando como parâmetro a instância do método de limpeza `self.my_cleanup` e qual parâmetro `my_cleanup` será chamado.

Repare que o erro continua sendo exibido. Caso ele seja ajustado, nossa ação de limpeza ainda será executada. O mais comum nesses métodos é fechar arquivos, *file descriptors*, conexões com bancos e outros recursos que precisam, explicitamente, ser fechados ou liberados.

8.10 Mocks/Stubs: outra abordagem de testes unitários

Até o momento, focamos em todos os nossos testes em código, que podemos chamar de *código de domínio*. O código de domínio é onde estão as regras que compõem o domínio no nosso programa, que, no caso, são dados. Porém, temos um pouco de código de *infraestrutura*, onde fizemos um download de um servidor remoto e depois extraímos o conteúdo do arquivo em uma pasta local. Essa parte do código é um pouco mais delicada de testar, pois dependemos, por exemplo, de uma conexão de internet para realizar um teste completo de download de um arquivo de um servidor remoto.

No nosso código de infraestrutura temos uma função que, da forma como foi feita, só poderia ser testada com um teste **funcional**. Essa função precisa de um objeto `response` que tem a resposta do servidor HTTP, com o con-

teúdo do arquivo cujo download queremos fazer. Se fosse criado um objeto de `response` explicitamente no caso de teste, e ele fosse passado para a função `download_length()`, estaríamos criando um *teste funcional* que executa em totalidade a funcionalidade, que é realizar o download de um arquivo.

Mas, em alguns casos, queremos testar código sem um ambiente funcional (sem conexão com internet, por exemplo) e de forma **isolada**. Existem algumas alternativas para o teste funcional. Veremos uma delas onde modificamos o comportamento de alguns objetos durante os testes. Com isso, vamos isolar o código testado de suas dependências. No final, teremos um teste unitário.

Para que isso fique mais claro, vamos analisar a nossa função de download e pensar sobre qual lógica queremos testar no seu código.

A lógica mais importante do código de download é a quantidade de vezes que ele chama os métodos `read()` e `write()`. Se o número de vezes em que esses métodos são chamados for correto, a função faz o download corretamente.

Se criamos *mocks* para os objetos `response` e `output`, podemos verificar quantas vezes os métodos foram chamados e com quais parâmetros. Com *mocks*, criaremos objetos que se comportam como queremos, e depois poderemos fazer perguntas a eles para testar se foram chamados da forma esperada. Assim, podemos testar a função `download_length` sem ter que criar um objeto real de `response` e sem depender de uma conexão com a internet.

Mocks versus Stubs

Tecnicamente, *mocks* e *stubs* são diferentes. Quando queremos apenas fazer com que determinados objetos tenham comportamentos preestabelecidos, o que precisamos são de *stubs*. Os *mocks* também podem ter comportamentos preestabelecidos, mas eles também têm expectativas a serem cumpridas.

Quando criamos um *mock* e trocamos uma dependência do código a ser testado por ele, queremos depois verificar se esse componente *mockado* foi chamado de maneira adequada. Desta forma, quando o criamos, é necessário configurar essas expectativas nele, pois a verificação delas fará parte do nosso

teste. Se você usar um *mock*, mas não configurar nem verificar nenhuma expectativa, você estará, na verdade, usando um *stub*. Ambos são comuns, mas no nosso caso, vamos focar um pouco mais nos *mocks*.

No Python 3.3, o projeto `mock` foi integrado dentro do módulo `unittest`. Logo, temos quase tudo o que precisamos já na biblioteca padrão.

Veja o exemplo a seguir:

```
import unittest
from unittest import mock

BUFF_SIZE = 1024

def download_length(response, output, length):
    times = length / BUFF_SIZE
    if length % BUFF_SIZE > 0:
        times += 1
    for time in range(int(times)):
        output.write(response.read(BUFF_SIZE))
        print("Downloaded %d" % (((time * BUFF_SIZE)/length)
                                *100))

class DownloadTest(unittest.TestCase):
    def test_download_with_known_length(self):
        response = mock.MagicMock()
        response.read = mock.MagicMock(side_effect=['Data']*2)

        output = mock.MagicMock()
        download_length(response, output, 1025)

        calls = [mock.call(BUFF_SIZE),
                 mock.call(BUFF_SIZE)]

        response.read.assert_has_calls(calls)

        calls = [mock.call('Data'),
                 mock.call('Data')]

        output.write.assert_has_calls(calls)
```

Podemos ver que existem duas etapas bem claras: configuração dos mocks e, depois do teste, as asserções. Na configuração do mock para response, primeiro criamos um MagicMock e, depois, outro MagicMock para o método sobre o qual queremos saber quantas vezes foi chamado.

O mock do método read() pode receber os valores que a chamada a ele retornará. No nosso caso, cada chamada retornará a string 'Data', portanto, o parâmetro side_effects pode ser uma lista com o valor repetido, como em ['Data', 'Data'] ou ['Data']*2.

Para o caso do objeto output, queremos também saber quantas vezes e com quais argumentos o método write foi chamado. Como não temos que configurar nenhum efeito colateral (*side effect*), basta criamos um mock para o objeto output.

Depois de a função ser executada com os mocks, conseguimos perguntar a eles sobre os métodos que foram executados. Como queremos verificar o número de vezes em que os métodos foram executados, usamos o método assert_has_calls(calls) do mock. Nesse caso, calls será uma lista de mock.call(arg). Repare que mock.call(arg) recebe um argumento que deve ser aquele com que o mock inicial foi chamado.

Por exemplo, vamos ver apenas um trecho:

```
response = mock.MagicMock()
response.read = mock.MagicMock(side_effect=['Data']*2)

download_length(response, output, 1025)

calls = [mock.call(BUFF_SIZE),
         mock.call(BUFF_SIZE)]

response.read.assert_has_calls(calls)
```

Esse trecho verifica se response.read() foi chamado duas vezes com o argumento BUFF_SIZE. Se calls tivesse mais um elemento, ou o argumento fosse diferente do passado para o mock, o teste geraria um erro.

Outro detalhe: o que é passado para output.write() é o que é retornado pelo mock de response.read(). Nesse caso, específicamos quais valores retornarão a cada chamada.

A mesma lógica vale para a asserção no `mock` de `output.write()`. Queremos verificar se ele foi chamado duas vezes, com o parâmetro `'Data'` em cada uma.

Outra função de download

Na função que não recebe o tamanho, a condição que termina a leitura é um retorno de `reponse.read()` com uma string vazia. Podemos usar novamente o atributo `side_effect` para ajudar com o teste. Veja o exemplo a seguir:

```
import unittest
from unittest import mock

BUFF_SIZE = 1024

def download(response, output):
    total_downloaded = 0
    while True:
        data = response.read(BUFF_SIZE)
        total_downloaded += len(data)
        if not data:
            break
        output.write(data)
        print('Downloaded {bytes}'.format(bytes=total_downloaded))


class DownloadTest(unittest.TestCase):
    def test_download_with_no_length(self):
        response = mock.MagicMock()
        response.read = mock.MagicMock(
                         side_effect=['data', 'more data', ''])

        output = mock.MagicMock()
        output.write = mock.MagicMock()

        download(response, output)

        calls = [mock.call(BUFF_SIZE),
```

```
                mock.call(BUFF_SIZE),
                mock.call(BUFF_SIZE)]

        response.read.assert_has_calls(calls)

        calls = [mock.call('data'),
                mock.call('more data')]

        output.write.assert_has_calls(calls)
```

Aqui, `read()` é executado três vezes, retornando uma string vazia na terceira vez e fazendo com que o `while` termine no comando `break`. Desta forma, `write()` é chamado duas vezes com os parâmetros `'data'` e `'more data'`.

Testar com `mocks` é uma técnica muito interessante em alguns casos, como das nossas funções de download. Felizmente, tudo o que precisamos está disponível na biblioteca padrão, no módulo `unittest` e `unittest.mock`.

8.11 Conclusão

Neste capítulo, aprendemos como criar casos de teste – herdando de `TestCase` –, vimos um pouco de sua API, como rodá-los por meio da API de linha de comando e como configurar programaticamente a nossa suíte de testes, além de um exemplo de uma customização que a API do pacote `unittest` permite.

Também foram vistos exemplos de testes que devem gerar erros, testes com `mocks` usando a própria biblioteca padrão e os métodos do ciclo de vida de um caso de teste: `setUp()` e `tearDown()`. Por último, aprendemos como criar ações de limpeza usando a própria API do `unittest`, ao usar `addCleanup()`.

No próximo capítulo, vamos ver o último conceito fundamental para trabalhar com Python, que são os módulos. Nosso contato com módulos até agora foi somente como *usuário*, ou seja, importando módulos e seus objetos, e usando-os. Agora, queremos aprender a criar módulos e entender melhor como funciona o seu mecanismo no Python 3.

Capítulo 9

Módulos e pacotes: organizando e distribuindo código

O último assunto dessa primeira parte do livro é *módulos*. Até aqui, o que vimos sobre módulos foi apenas como importá-los e exemplos de como importá-los da biblioteca padrão.

Agora, vamos ver como criar módulos e como utilizar os que criamos. Com esse recurso, poderemos compor nosso programa em diversos módulos. A modularização é um dos princípios básicos do universo de desenvolvimento de software.

Como já falamos em testes, fica claro que, até então, temos dois *tipos* de código: domínio e testes. É natural e comum ter um módulo para cada um desses tipos. Também é comum, mas nem sempre necessário, existir um módulo de testes para cada módulo de domínio.

O nosso domínio é simples: tabelas de dados, colunas e relacionamentos. Na prática, temos quatro classes de domínio. Como são poucas, vamos colocar todas no módulo `domain.py`. Além deste, teremos um módulo de testes chamado `test_domain.py`, onde colocaremos os testes que fizemos.

Em projetos maiores, podemos perceber estilos de modularização, como maior ou menor número de módulos. Alguns desenvolvedores pensam em termos de grupos mais generalistas, enquanto outros vão modularizar seu programa sob uma perspectiva mais especialista.

Agora, vamos olhar para a modularização de forma prática.

Exemplos reais

Um dos principais motivos da modularização chama-se **reúso**.

Das bibliotecas disponíveis junto ao interpretador, a famosa biblioteca padrão (ou *standard library*), nós importamos módulos e objetos dos módulos para utilizá-los em nossos programas. Nesses casos, usamos os comandos de `import`. Perceba que acabamos de reusar código já pronto. Certamente, você encontrará módulos úteis na biblioteca padrão e em diversos outros projetos de código aberto.

Também existem casos nos quais criamos módulos com determinados nomes e ferramentas que importam e interagem com o que criamos nesses módulos. Isso é muito comum em frameworks de desenvolvimento web. Aqui, não exploraremos explicitamente o reúso, mas vamos usufruir do fato de que um framework pode importar um módulo nosso e usá-lo.

Vamos entender como módulos são criados e importados, para que sua utilização seja feita de forma adequada em ambos os casos.

Criando módulos: arquivos .py

Quando criamos o arquivo `domain.py` no capítulo anterior, criamos um módulo. Todo arquivo `.py` que contém código válido e está no *path* de busca de módulos pode ser carregado.

Caminho (*path*) de busca de módulos

Em Python, quando importamos um módulo, o interpretador percorre uma série de caminhos – os chamados *paths* –, para procurar a implementação do módulo importado. Por padrão, o diretório de trabalho, de onde o interpretador foi chamado, está nessa lista. Por isso, quando criamos um arquivo `meu_modulo.py`, se abrimos o interpretador no mesmo diretório dele, podemos executar com sucesso `import meu_modulo` e utilizar o seu código.

Como esse arquivo está na raiz do diretório de trabalho, podemos importá-lo usando o comando `import` no *path* de busca padrão. O conteúdo do módulo `domain` contém as quatro classes que definimos no capítulo 8: `DataTable`, `Column`, `PrimaryKey` e `Relationship`.

Veja o exemplo:

```
>>> import domain
>>> table = domain.DataTable("Empreendimento")
>>> table
<domain.DataTable object at 0x10c7f7fd0>
```

No capítulo 3 aprendemos a usar esse comando. A única diferença agora é que estamos importando do nosso próprio arquivo `domain.py`.

Outra melhoria que podemos fazer é reunir todos os testes no arquivo `test_domain.py`. O prefixo `test_` faz com que o buscador de testes do `unittest` consiga achar esse módulo de testes e executar os casos de testes que encontrar, dentro dele.

9.1 Módulos em Python: primeiro passo

Sempre que criamos um arquivo `*.py`, criamos um módulo. Este pode ser importado usando o comando `import nome_modulo`. Sempre que um novo código for criado, devemos avaliar se ele precisa de um módulo novo ou não. Caso precise, basta criar um novo arquivo `*.py`.

Revisando o modelo de classes e seus testes

Agora vamos aproveitar e revisar os arquivos de trabalho: `domain.py` e `test_domain.py`.

Removi os *docstrings* das classes de domínio para não ocupar muito espaço.

Arquivo `domain.py`:

```
import decimal

class Column:
    def __init__(self, name, kind, description=""):
        self._name = name
        self._kind = kind
        self._description = description
        self._is_pk = False

    def __str__(self):
        _str = "Col: {} : {} {}".format(self._name,
                                        self._kind,
                                        self._description)
        if self._is_pk:
            _str = "({}) {}".format("PK", _str)
        return _str

    @staticmethod
    def validate(kind, data):
        if kind == 'bigint':
            if isinstance(data, int):
                return True
            return False
        elif kind == 'varchar':
            if isinstance(data, str):
                return True
            return False
        elif kind == 'numeric':
            try:
                val = decimal.Decimal(data)
            except:
```

```
                return False
            return True


class PrimaryKey(Column):
    def __init__(self, table, name, kind, description=None):
        super().__init__(name, kind, description=description)
        self._is_pk = True


class Relationship:
    def __init__(self, name, _from, to, on):
        self._name = name
        self._from = _from
        self._to = to
        self._on = on


class DataTable:
    def __init__(self, name):
        self._name = name
        self._columns = []
        self._references = []
        self._referenced = []
        self._data = []

    def _get_name(self):
        return self._name

    def _set_name(self, _name):
        self._name = _name

    def _del_name(self):
        raise AttributeError("Não pode deletar esse atributo")

    name = property(_get_name, _set_name, _del_name)
    references = property(lambda self: self._references)
    referenced = property(lambda self: self._referenced)
```

```
    def add_column(self, name, kind, description=""):
        self._validate_kind(kind)
        column = Column(name, kind, description=description)
        self._columns.append(column)
        return column

    def _validate_kind(self, kind):
        if not kind in ('bigint', 'numeric', 'varchar'):
            raise Exception("Tipo inválido")

    def add_references(self, name, to, on):
        relationship = Relationship(name, self, to, on)
        self._references.append(relationship)

    def add_referenced(self, name, by, on):
        relationship = Relationship(name, by, self, on)
        self._referenced.append(relationship)
```

No nosso aplicativo, o módulo `domain` contém as classes do domínio da nossa aplicação. Em um projeto com diversos módulos, geralmente existe um módulo de testes para cada um, com prefixo `test_` (por exemplo, `test_domain.py`).

Veja o código feito até então de `test_domain.py`:

```
import unittest

from domain import DataTable

class DataTableTest2(unittest.TestCase):
    def setUp(self):
        self.table = DataTable('A')

    def test_add_column(self):
        self.assertEqual(0, len(self.table._columns))

        self.table.add_column('BId', 'bigint')
        self.assertEqual(1, len(self.table._columns))

        self.table.add_column('value', 'numeric')
```

```
        self.assertEqual(2, len(self.table._columns))

        self.table.add_column('desc', 'varchar')
        self.assertEqual(3, len(self.table._columns))

    def test_add_column_invalid_type(self):
        self.assertRaises(Exception, self.table.add_column,
                          ('col', 'invalid'))

    def test_add_column_invalid_type_fail(self):
        a_table = DataTable('A')
        error = False

        try:
            a_table.add_column('col', 'invalid')
        except:
            error = True

        if not error:
            self.fail("Chamada não gerou erro mas deveria")

    def test_add_relationship(self):
        a_table = DataTable('A')
        col = a_table.add_column('BId', 'bigint')
        b_table = DataTable('B')
        b_table.add_column('BId', 'bigint')
        a_table.add_references('B', b_table, col)

        self.assertEqual(1, len(a_table.references))
        self.assertEqual(0, len(a_table.referenced))

    def test_add_reverse_relationship(self):
        a_table = DataTable('A')
        col = a_table.add_column('BId', 'bigint')
        b_table = DataTable('B')
        col = b_table.add_column('BId', 'bigint')
        b_table.add_referenced('A', a_table, col)

        self.assertEqual(1, len(b_table.referenced))
```

```
        self.assertEqual(0, len(b_table.references))
```

9.2 O que acontece quando importamos um módulo?

Quando o módulo é importado, todos os comandos nele são executados. As definições de classes e funções ficam em uma tabela de símbolos do próprio módulo importado, evitando conflitos com definições de outros módulos.

Esse módulo `domain` possui quatro classes. Quando ele é importado, esses quatro objetos serão acessíveis de alguma forma. O que nos interessa nele são as classes que este define. Existem várias formas de acessá-las. Vamos ver como usar esse módulo que criamos e também as variações.

Importação simples

A forma mais simples de importar é usar o comando `import`, passando apenas o(s) nome(s) do(s) módulo(s).

```
>>> import domain
```

Agora, no código que importou, a variável (ou nome) `domain` representa o módulo e nos dá acesso às classes.

```
>>> import domain
>>> table = domain.DataTable("Empreendimento")
```

Uma característica que fica explícita é o uso do código do módulo importado, já que sempre temos que usar o "prefixo" `domain.` antes de utilizar alguma definição do módulo.

Na variação com vários nomes, eles são separados por vírgula:

```
import os, sys
```

Importação composta com nomes

A importação composta nada mais é que o comando de `import` com duas informações: o nome do módulo **e** o nome da definição. A sintaxe é `from nome_modulo import nome_definicao`.

```
>>> from domain import DataTable
>>> DataTable
<class 'domain.DataTable'>
```

O nome da definição pode ser um *wildcard* (*) que importa **todos** os nomes do módulo. Nem sempre queremos importar tudo, mas isso pode ser conveniente em explorações no console.

```
>>> from domain import *
>>> table = DataTable("Empreendimento")
>>> col = Column("IdEmpreendimento", 'bigint')
```

Aqui, também podemos separar as definições por vírgula:

```
>>> from domain import DataTable, Column
>>> table = DataTable("Empreendimento")
>>> col = Column("IdEmpreendimento", 'bigint')
```

Em alguns casos, quando importamos algumas definições e os nomes ocupam muito espaço, exigindo uma quebra de linha, podemos usar a sintaxe com tuplas:

```
>>> from domain import (DataTable, Column, PrimaryKey,
...                     Relationship)
```

Modificando nomes importados localmente

Pode ser necessário, ou apenas conveniente, que o nome da definição (ou do módulo) que queremos importar seja diferente no módulo que está importando. Isso é possível usando as variações com a palavra reservada `as`.

```
>>> from domain import DataTable as Table
>>> table = Table("Empreendimento")
```

Veja dois exemplos no próprio código do aplicativo:

```
>>> import unittest.mock as mock
>>> import urllib.request as request
```

Aqui, não existem regras de certo ou errado. Mas lembrando o *The Zen of Python*, visto no capítulo 1: *explícito é melhor que implícito*, e muita coisa implícita pode gerar confusão. Nesse caso, o objetivo é deixar explícito que um código importado está sendo usado.

9.3 Pacotes: agrupando módulos

Projetos mais complexos não são organizados apenas em módulos. Quando a quantidade de módulos aumenta, aumenta também a necessidade de colocarmos alguns em pastas, que reúnem módulos de um determinado tipo.

No nosso projeto, temos o código de infraestrutura (download, extração de `zip`, leitura de arquivos) e domínio (`DataTable`, `Column` etc.). Nosso próximo passo agora é organizar nossa aplicação em pacotes.

Primeiro, vamos criar um pacote no nível raiz. Nele, colocaremos o módulo `domain` e o subpacote `commands`. Nesse subpacote, colocaremos os módulos que compõe as etapas de tratamento dos dados, do download e a geração do nosso modelo de tabelas.

A estrutura ficaria semelhante à seguinte:

```
| copa_transparente
+ | __init__.py
  | domain.py
  | commands/
  | +| __init__.py
      | read_meta.py
      | download_data.py
      | extract_data.py
```

Para que o interpretador reconheça pastas como pacotes, ele precisa encontrar o arquivo `__init__.py`, mesmo que vazio, dentro da pasta. Agora, podemos chamar o nosso projeto de pacote. O pacote `copa_transparente`.

Isso faz com que `import domain` não funcione mais. De agora em diante, nosso código está acessível com o prefixo `copa_transparente`. O código anterior agora deve ser escrito da seguinte forma:

```
from copa_transparente.domain import DataTable
```

Apesar de inicialmente parecer que temos um trabalho a mais, na hora de importar, os arquivos `__init__.py` servem para que seja feito o controle fino de quais nomes são exportados quando o pacote for carregado.

Se executarmos `import copa_transparente`, o arquivo `__init__.py` na raiz do diretório `copa_transparente` será executado. Todas as definições desse módulo e as importadas por ele ficarão disponíveis para o código cliente. Entretanto, isso nem sempre é o que queremos. No nosso aplicativo, por enquanto, a única classe que é de interesse ao usuário final é `DataTable`. Todas as outras devem ser escondidas do usuário, para diminuir a complexidade da manipulação do projeto.

9.4 Definindo nomes para exportação e escopo de exportação

Em alguns projetos, não queremos expor para o usuário a separação conceitual interna do nosso código. Para contornar isso, podemos controlar quais nomes serão exportados por um módulo, mesmo que estes não estejam originalmente declarados nele. O interessante disso é que deixamos a critério do usuário a escolha de como importar o que ele quer: fazendo referência ao caminho inteiro, ou apenas importando o nome.

Por exemplo, no nosso projeto, não queremos que o usuário conheça o módulo `domain`, mas queremos que ele use a classe `DataTable` que está em `copa_transparente.domain`. Na prática, queremos trocar isto:

```
>>> from copa_transparente.domain import DataTable
```

Por isto:

```
>>> from copa_transparente import DataTable
```

Esse tipo de decisão faz parte do **design** da API. No nosso caso, como não expusemos muitos objetos, é mais prático colocar todos debaixo de um único nome – no caso, `copa_transparente` – em vez de obrigar o usuário a fazer diversos `import`s de submódulos distintos.

Sempre que um módulo é importado, a variável `__all__` é procurada dentro dele. Caso exista, ela que definirá quais são os nomes exportados pelo módulo. Caso não exista, **todas** as definições encontradas no módulo que não começam com `_` serão exportadas.

No nosso caso, o primeiro exemplo é de como exportar apenas um nome, sem definir a variável `__all__`. No arquivo `__init__.py`, dentro da pasta `copa_transparente`, podemos ter o seguinte código:

```
>>> from domain import DataTable
```

Repare que aqui estamos no `__init__.py`, dentro da pasta `copa_transparente`, e não como código de usuário. Portanto, não precisamos do "prefixo" `copa_transparente`. Esse código anterior permite que tenhamos, no nosso código cliente:

```
>>> from copa_transparente import DataTable
```

O que aconteceu aqui é que, quando foi feito o `import` de `copa_transparente`, o conteúdo de `copa_transparente.__init__` foi executado, e o nome `DataTable` foi importado do módulo `domain` e automaticamente exportado pelos clientes de `copa_transparente`. Isso porque tudo que está definido no `__init__.py` de um módulo é exportado, por padrão.

Em alguns momentos de projetos mais complexos, queremos (ou precisamos) importar diversas classes no `__init__.py`, mas não queremos expô-las para os clientes do módulo. É aí que entra a variável `__all__.py`. Usando-a, poderíamos fazer no arquivo `__init__.py` (na raiz da pasta `copa_transparente`), da seguinte forma:

```
from domain import *

__all__ = ['DataTable']
```

O efeito prático é idêntico para o nosso exemplo. A diferença é que, no primeiro caso, não temos acesso, por exemplo, à classe `Column` no escopo do módulo `__init__`. Já no segundo exemplo, temos `Column` no escopo do módulo `__init__` e acesso a todas as classes definidas em `domain`, e optamos por exportar apenas `DataTable`.

9.5 Conclusão

Neste capítulo, vimos como criar nossos módulos e alguns exemplos práticos de como utilizar o mecanismo de módulos, tanto do ponto de vista de um autor de uma biblioteca quanto pelo de um usuário. Vimos também o conceito de pacote, e como pacotes e módulos relacionam-se. Por último, aprendemos como usar as variações do comando de `import` e como customizar que nomes são exportados por nossos módulos.

De uma forma geral, o que vimos aqui cobre 90% do uso diário de módulos, tanto como criadores de módulos e pacotes assim como usuários de módulos de terceiros.

A seguir, vamos aprender a manipular arquivos, para permitir a leitura de dados que populará o modelo de objetos que criamos.

Parte II

Mundo Python – Além dos recursos fundamentais

Na parte I, vimos alguns elementos básicos de uma linguagem de programação e como eles funcionam em Python.

O que iniciaremos agora é a exploração de diversos aspectos, além dos recursos fundamentais. Vamos ver como trabalhar com arquivos – já que nosso programa vai ler arquivos de dados –, e algumas classes de tipos de dados como decimais, datas e conjuntos, que estão disponíveis na linguagem. Além disso, também apresentarei novos elementos de sintaxe, aspectos mais avançados do modelo de objetos e outras coisas que fazem parte da linguagem e merecem estar no livro.

Capítulo 10

Trabalhando com arquivos

Neste capítulo, aprenderemos a manipular arquivos, para então podermos ler os arquivos de dados da base que estamos usando para popular nosso modelo de objetos, desenvolvido no capítulo 7.

10.1 Importando dados para as tabelas

Até o momento, criamos algumas classes que serão a base do nosso aplicativo, mas elas ainda não estão nos permitindo muitas coisas. O que queremos agora é começar a ler os arquivos de dados e a criar as instâncias dessas tabelas, usando os dados das bases das quais podemos fazer download. Depois que as tabelas foram criadas e os dados dos arquivos lidos, o objetivo é usá-las para realizar consultas nos dados.

Para ler arquivos, temos a função *builtin* (embutida), chamada `open()`, que abre um arquivo e retorna um objeto do tipo arquivo. No Python 3, exis-

tem três tipos de arquivos: binários, binários bufferizados e de texto. O arquivo que vamos ler é um arquivo texto. Vamos começar com o exemplo mais simples possível e aproveitar para ver que tipo tem um objeto arquivo quando aberto com os parâmetros padrão:

```
>>> data = open('data/data/ExecucaoFinanceira.csv', 'r')
>>> data
<_io.TextIOWrapper name='ExecucaoFinanceira.csv'
mode='r' encoding='UTF-8'>
>>> data.close()
```

O código anterior passa como parâmetro o caminho e `'r'` que é o modo, no caso `read` ou leitura, que serve para abrir o arquivo para leitura dos dados. Quando mencionamos objeto do tipo arquivo, a ideia é que, independente de seu tipo, sua interface seja igual para todos. Mesmo que Python não tenha interfaces como Java, os objetos ainda podem seguir determinadas padronizações de métodos, ou até mesmo herdar de classes abstratas.

A assinatura completa da função é: `open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)`. Os modos informam se o arquivo é texto ou binário, e para quais operações será aberto (leitura, escrita etc.).

Veja a lista de modos a seguir:

```
'r'    open for reading (default)
'w'    open for writing, truncating the file first
'x'    open for exclusive creation, failing if the file already
exists
'a'    open for writing, appending to the end of the file if it
exists
'b'    binary mode
't'    text mode (default)
'+'    open a disk file for updating (reading and writing)
```

Vale lembrar que os modos podem ser combinados, se pertinente, como em: `wb`, sendo escrita binária (*write binary*); ou `rt`, como leitura texto (*read text*).

O parâmetro `buffering` permite algumas opções: `-1` é o valor padrão, que significa usar as configurações do sistema; `0` para não usar buffer (so-

mente no modo binário); `1` para usar um buffer por linhas (válido somente em arquivo texto); e, em valores > 1, o tamanho do buffer passa a ser o valor passado como parâmetro.

O parâmetro `encoding` pode ser a string com o nome do formato, como `"utf-8"`.

> Os outros parâmetros são muito específicos e podem ser consultados na documentação oficial, em https://docs.python.org/3/library/functions.html#open.

Voltando ao exemplo, veja que o objeto retornado é do tipo `TextIOWrapper`. Essa classe é uma implementação de arquivos de texto, que, além de ter as funções comuns a todo tipo de objeto do tipo arquivo, tem também funções específicas que só podem ser usadas para arquivos abertos em modo texto.

Os tipos de arquivo de texto trabalham com strings em vez de bytes, como nos arquivos binários. No capítulo 2, vimos que no Python 3 todas strings são **unicode** e, portanto, quando escritas em arquivo ou enviadas pela rede, devem ser convertidas para bytes através de um *encoder*. Nos tipos de arquivo de texto, mecanismos internos já fazem a conversão de strings para bytes (e vice-versa).

Como queremos olhar para os tipos de arquivo de forma mais genérica, vamos nos concentrar nos métodos em comum a todos os tipos de arquivo, e ver a seguir dois dos mais importantes: `read()` e `write()`.

10.2 Lendo arquivos

A função para ler dados é `read()`. Quando a invocamos sem parâmetros, todo o conteúdo do arquivo é lido e retornado. Se o arquivo for muito grande, o seu programa pode consumir muita memória. Uma alternativa é especificar a quantidade de bytes a ser lida, como `read(4096)`. Vamos ver o exemplo:

```
>>> data = open('ExecucaoFinanceira.csv', 'r')
>>> data.read(19)
```

```
'1;2;132;CONSTRUTORA'
>>> data.close()
```

O código exibiu os 19 primeiros bytes do arquivo de execuções financeiras. Como estamos trabalhando com arquivos de texto que seguem o formato CSV, geralmente as quebras de linha `"\n"` delimitam o final de uma informação. Por exemplo, no caso de um arquivo de dados `copa_transparente` [1], cada linha significa um registro naquela tabela. Logo, uma linha no arquivo `ExecucaoFinanceira.csv` contém a informação de uma Execução Financeira.

Existe uma forma melhor de ler, linha a linha, um arquivo de texto, que veremos na sequência.

Iterando nas linhas

No nosso aplicativo, como todos os arquivos têm menos de 10Mb, podemos lê-los de uma vez só, apenas chamando `read()`, sem parâmetro algum. Em nosso caso, estamos lidando com um CSV, então queremos ler linha a linha. Em Python, podemos usar o método `readline()` ou iterar as linhas do arquivo, usando o comando `for` no próprio objeto do tipo arquivo. Veja o exemplo:

```
data = open('data/data/ExecucaoFinanceira.csv', 'r')
for line in data:
    print(line)
data.close()
```

O método `readline()` está disponível apenas para arquivos de texto. A iteração usando o comando `for` também funciona em arquivos binários, mas continuará fazendo a quebra pelo caractere `"\n"`.

Se quisermos todas as linhas em uma lista de strings, em que cada item da lista é uma linha do arquivo, podemos usar a função `readlines()`. A desvantagem é que, novamente, em arquivos grandes, pode ser criada uma lista muito grande em memória. Deve-se analisar caso a caso se é necessário ter tudo em memória simultaneamente, ou se ter acesso apenas a uma linha por vez já atende à necessidade.

Para ver um exemplo da função `readlines()`, vamos ver mais um tipo de arquivo de texto que é muito comum: `StringIO`. Um `StringIO` é um *stream* de texto em memória, que implementa o contrato da API de manipulação de arquivos de texto. É muito útil quando queremos usar um código que espera um objeto do tipo arquivo e nós podemos passar um objeto compatível, mas que por dentro tem uma string que é, na verdade, o conteúdo do arquivo.

```
>>> import io
>>> data = io.StringIO("a\nb\nc")
>>> lines = data.readlines()
>>> print(lines)
['a\n', 'b\n', 'c']
>>> data.close()
>>> data.closed
True
```

Para o código, `data` é um arquivo e pode ser usado como arquivo em códigos que esperam um tipo arquivo texto.

Como esperado, a função `readlines` retornou uma lista com cada linha do arquivo. Repare que as linhas que têm o caractere `"\n"` vêm com ele na string, e as linhas que não têm (no caso, apenas a última), vêm sem.

ITERAR *VERSUS* LER TODAS AS LINHAS

A vantagem de iterar nas linhas é que em nenhum momento precisamos ter em memória um espaço grande o suficiente para caber todo o conteúdo. Para arquivos muito grandes, isso pode fazer uma boa diferença.

Em alguma situações, pode ser desejável ter o conteúdo todo em memória. Então, cada situação deve ser avaliada separadamente.

10.3 Fechando arquivos

Programas "bem comportados" fecham os arquivos abertos ao final de sua execução. Nos nossos exemplos, fizemos isso usando o método `close()`.

Todos os tipos de arquivo implementam o método `close()`, até mesmo `StringIO`, que não tem um arquivo real aberto e que, depois do método `close()` chamado, seu conteúdo não pode ser mais lido. Nesses casos, a chamada ao método `read()`, em um arquivo fechado, gera um `ValueError`.

Podemos verificar se um objeto do tipo arquivo está fechado usando a propriedade `closed`. Veja o exemplo que ilustra o que falamos:

```
>>> import io
>>> data = io.StringIO("a\nb\nc")
>>> lines = data.readlines()
>>> print(lines)
['a\n', 'b\n', 'c']
>>> data.close()
>>> data.closed
True
>>> data.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

10.4 Abrindo arquivos com o comando with

Em Python, existe uma forma **pythônica** de se trabalhar com arquivos usando o comando `with`. Esse comando introduzido pela PEP-343 [4] tem o objetivo de simplificar o uso repetido de comandos `try/finally` em algumas situações. Conceitualmente, temos um gerenciador de contexto, que é uma classe de cujo ciclo de vida dois métodos fazem. Por meio desses métodos, conseguimos executar código em momentos como entrada e saída de um bloco com `with`.

Com esse comando, não precisamos nos preocupar em fechar explicitamente o arquivo, já que o próprio gerenciador de contexto será chamado no final do bloco `with`, e chamará o método `close()` do arquivo em questão. Vamos usá-lo no nosso primeiro exemplo:

```
with open('data/data/ExecucaoFinanceira.csv', 'r') as data:
    content = data.read()
    print(content)
```

Nesse exemplo, o objeto retornado pela função `open()` é compatível com o protocolo de gerenciador de contexto. Portanto, quando o bloco `with` termina, o arquivo retornado por `open('ExecucaoFinanceira.csv', 'r')` é fechado. Algumas bibliotecas de acesso a banco de dados usam esse comando para delimitar conexões abertas, ou até mesmo transações em bancos de dados.

Se juntarmos o comando `with` com leitura de arquivo linha a linha e *split* de strings, já conseguimos obter todos os valores de execuções financeiras da base de dados da copa. Veja o exemplo:

```
with open('data/data/ExecucaoFinanceira.csv', 'r') as data:
    for line in data:
        print(line.split(';')[5])
```

Veremos na saída valores de execuções financeiras até que um erro seja exibido quando uma linha mal formada for encontrada. Por enquanto isso não é um problema já que o foco é sobre abrir um arquivo, ler linha a linha e deixar que o próprio ambiente se encarregue de fecha-lo ao final do uso.

10.5 Escrevendo em arquivos

No nosso primeiro programa, no capítulo 4, vimos um exemplo de escrita em arquivo no código que faz o download do arquivo da base de dados. A escrita no arquivo se deu basicamente em duas linhas do código. Vamos analisar:

```
>>> out_file = io.FileIO(file_path, mode="w")
```

Veja o código anterior usando a função `open()`:

```
>>> out_file = open(file_path, mode="wb") # modo write binary
```

Agora, as chamadas da função `write()`:

```
>>> out_file.write(bytes_to_write)
```

Por se tratar de um arquivo `ZIP` – que tem conteúdo binário –, usamos a classe `FileIO` que trabalha com bytes em vez de strings. Porém, poderíamos usar a função `open()`, sinalizando arquivo binário, exatamente como no

exemplo anterior. Além disso, abrimos o arquivo com modo w, que é o modo de escrita (*write*), e que permite que a função write() seja chamada. O código completo não faz nada além de ler dados da resposta do servidor e escrever em um arquivo local.

Se estivéssemos trabalhando com arquivos texto, a única diferença seria o método write(), que estaria esperando uma string (e não bytes), como no nosso exemplo. É muito importante escolher o tipo correto dependendo do arquivo que será lido, pois, caso contrário, as coisas podem não funcionar como esperado.

10.6 Navegação avançada com seek()

Uma função base de arquivos é a seek(). Essa função é muito semelhante à fseek() da linguagem C. Com ela, você pode colocar o ponteiro do arquivo no lugar que desejar. Isso é muito útil quando queremos ler o mesmo trecho de um arquivo diversas vezes.

Veja o exemplo:

```
>>> data = open('ExecucaoFinanceira.csv', 'r')
>>> data.read(5)
'1;2;1'
>>> data.read(5)
'32;CO'
>>> data.read(5)
'NSTRU'
>>> data.read(5)
'TORA '
>>> data.seek(0)
0
>>> data.read(20)
'1;2;132;CONSTRUTORA '
```

Repare que, a cada chamada de read(), o ponteiro do arquivo é atualizado e a nova leitura é iniciada em uma outra região. Por meio do uso da função fseek(), foi possível retornar ao início do arquivo para realizar a leitura.

10.7 Conclusão

Neste capítulo, vimos como abrir e fechar arquivos, e como usar o comando `with` para que eles sejam fechados automaticamente. Depois aprendemos algumas formas de se obter os dados dos arquivos, seja lendo uma quantidade específica de bytes, iterando linha a linha com o loop `for`, ou até mesmo pegando uma lista com todas as linhas do arquivo.

Além disso, vimos também como usar os modos de abertura, interferindo nos objetos retornados, e quais operações podemos realizar neles. Por fim, aprendemos como escrever em arquivos e em quais detalhes devemos ficar atentos. A seguir, veremos mais recursos disponíveis na biblioteca padrão, mas que não estão embutidos na linguagem e precisamos importá-los.

Capítulo 11

Um passeio por alguns tipos definidos na biblioteca padrão

Até agora, vimos alguns dos tipos que podem ser chamados **básicos**, que são: inteiros, flutuantes e strings. Os tipos que chamamos de básicos são os que não necessitam de nenhum comando de `import` e já estão disponíveis no contexto de execução. Esses tipos também são chamados de **builtins**, pois estão embutidos na própria linguagem e podem ser usados sem que nenhum módulo seja importado.

Também vimos as estruturas de dados **fundamentais**, que são tuplas, listas e dicionários. Podemos chamá-las de fundamentais, porque toda linguagem é implementada usando-as como base.

No mundo real, em algumas situações, precisamos de outros recursos como um tipo mais especializado, ou uma estrutura de dados que não sejam

as fundamentais, para resolvermos nossos problemas com mais eficiência.

Em Python, existem muitos recursos que estão disponíveis na biblioteca padrão, mas que não estão embutidos na linguagem. Isso significa que podemos usá-los apenas importando de onde quer que seja. No caso do nosso aplicativo, queremos um tipo que nos permita trabalhar com valores financeiros e que suportem operações aritméticas com precisão arbitrária. Em muitas linguagens, esse tipo é chamado, informalmente, de Decimal, como por exemplo em Python, no qual temos o tipo `Decimal`, ou em Java, onde existe o `java.math.BigDecimal`.

11.1 Dinheiro e decimais com precisão arbitrária

Quando estamos realizando cálculos financeiros, mesmo que simples, precisamos de números com precisão arbitrária. Quando usamos flutuantes (**floats**) para representar valores fracionários – como `1.25` –, dependendo da operação que fizermos, o resultado dos arredondamentos pode causar muita surpresa. Veja os exemplos a seguir:

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') -
                                                Decimal('0.3')
Decimal('0.0')
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') -
                                                Decimal('0.3')
Decimal('0.0')
```

Acontece que os flutuantes são uma forma de representar *aproximações* de números reais para um grande intervalo de números. Para termos essa flexibilidade de suportar intervalos realmente grandes, sua precisão é um pouco sacrificada. Em muitos domínios, essa perda de precisão não afeta o resultado do programa, então, escolher um dos dois é uma questão de design do projeto.

No nosso programa, vamos trabalhar com valores que representam quantias em dinheiro. Na nossa moeda, temos duas casas decimais, e queremos precisão total e arredondamentos coerentes com as contas financeiras. Para esse tipo de tarefa, usamos o tipo `Decimal` do módulo `decimal`.

Tudo o que precisamos fazer é converter os valores financeiros de string para `Decimal`, para poder realizar as contas que nos interessam.

Vamos voltar no exemplo do capítulo anterior e utilizar o `Decimal`, para que seja possível somar todos os valores com a precisão que queremos.

```
# coding: utf-8

from decimal import Decimal

total = Decimal('0')

with open('data/data/ExecucaoFinanceira.csv', 'r') as data:
    for line in data:
        try:
            info = line.split(';')
            str_value = info[5] # o valor está na 5a posição
            total += Decimal(str_value)
        except Exception as e:
            print('error {}'.format(line))

print("Total gasto: {}".format(total))
```

O resultado computado foi de:

```
Total gasto: 22583122032.97
```

No código anterior, criamos os decimais a partir de strings, e os manipulamos de forma idêntica ao que fazemos com inteiros ou flutuantes. Eles têm algumas funções matemáticas disponíveis e, de maneira geral, são intercambiáveis com os tipos primitivos.

Mais detalhes em Decimals

Um erro muito comum é em relação à inicialização dos decimais, mais especificamente no tipo que passamos no construtor. O construtor aceita

strings, inteiros, floats e tuplas. Vamos ver alguns casos que expressam melhor esses erros que podem ser cometidos.

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1')
Decimal('0.2')
>>> Decimal(0.1) + Decimal(0.1)
Decimal('0.2000000000000000111022302463')
>>> Decimal(0.1) + Decimal(0.1) ==
        Decimal('0.1') + Decimal('0.1')
False
```

Como mencionado, os **floats** são aproximações. Logo, quando passamos para um construtor do tipo decimal, ele capta a aproximação, e não o número representado pelo literal `0.1`.

É possível configurar a precisão nas operações, para que a última igualdade do exemplo anterior seja verdadeira. Fazemos isso configurando a precisão no contexto dos decimais. Veja o exemplo a seguir:

```
>>> from decimal import Decimal
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 1
>>> Decimal(0.1) + Decimal(0.1) ==
        Decimal('0.1') + Decimal('0.1')
True
```

Se a precição exigida é de apenas uma casa decimal, então, a igualdade anterior tem de ser verdadeira.

Também são aceitos valores string que representam um valor que não seja um número (*Not a Number*), zero negativo ou infinitos. Veja os exemplos a seguir:

```
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('Infinity')
Decimal('Infinity')
>>> Decimal('-Infinity')
Decimal('-Infinity')
>>> Decimal('-0')
Decimal('-0')
```

Impedindo uso de decimais com certos tipos

A última característica que vamos ver nos decimais é como usar o seu mecanismo de armadilhas (*traps*). Com uma armadilha, podemos solicitar que uma exceção seja levantada caso um decimal seja usado em uma operação de construção, ou comparação com um outro valor que não é um decimal.

```
>>> c = getcontext()
>>> from decimal import FloatOperation
>>> c.traps[FloatOperation] = True
>>> Decimal('1.10') < 1.25
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
```

Podemos usar esse mecanismo para emitir, em tempo de execução, uma exceção quando uma operação envolvendo um decimal e um objeto de outro tipo, que não decimal, for executada.

No nosso aplicativo, o que precisamos é apenas do tipo `Decimal`, para realizar algumas operações aritméticas. É importante tratar erros que possam ser gerados na chamada do construtor, e entender qual motivo os levou a acontecer, para tomar a melhor decisão sobre o tratamento dessa situação.

11.2 Datas e tempo: módulo datetime

Agora que sabemos como tratar corretamente os valores financeiros, podemos criar consultas mais interessantes, usando datas como referência. Por exemplo, podemos saber quais execuções financeiras foram assinadas a partir de determinada data, ou em um intervalo específico de tempo.

Em Python, a manipulação de datas faz uma separação simples de início: trabalhar com datas e hora, trabalhar somente com data, ou trabalhar somente com hora (ou horário). Em cada caso, uma classe específica deve ser importada, que pode ser: `date`, `datetime` ou `time`, todas do pacote `datetime`.

Vamos supor que nosso objetivo seja ver as execuções financeiras assinadas entre duas datas. Precisamos transformar o input de texto em objetos

de data, e compará-los aos objetos criados arbitrariamente, por exemplo. A seguir, veja um programa que calcula o total assinado entre 2009 e 2010.

```
# coding: utf-8

from decimal import Decimal
from datetime import date

total = Decimal('0')
start_date = date(2009, 1, 1)
end_date = date(2010, 1, 1)

def get_value(info):
    try:
        signature_str_date = info[7]
        year = int(signature_str_date.split('/')[2])
        month = int(signature_str_date.split('/')[1])
        day = int(signature_str_date.split('/')[0])
        signature_date = date(year, month, day)

        if signature_date > start_date and signature_date <
        end_date:
            str_value = info[5]
            value = Decimal(str_value)
            return value
    except Exception as e:
        print(e)
        pass
    return Decimal('0')

with open('data/data/ExecucaoFinanceira.csv', 'r') as data:
    for line in data:
        total += get_value(line.split(';'))

print("Total gasto com assinaturas entre {} e {} : {}"
    .format(start_date, end_date, total))
```

Primeiro, usamos o construtor de `date` com a seguinte assinatura: `date(year, month, day)`. Com ele, criamos as datas de referência. Já

as datas das execuções criamos a partir das datas em texto, de cada linha do arquivo.

No final, realizamos a comparação com o operador >, que permite comparação de valores de data, data e hora, e hora. Todas as comparações estão disponíveis para os objetos de data, desde que sejam compatíveis entre si.

O objeto `date` é o mais simples de todos e conta apenas com três atributos: ano, mês e dia. Na prática, a maioria dos programas acaba usando o objeto `datetime`, que é mais completo e tem mais funcionalidades, além de poder ser usado mesmo que as informações de hora não sejam especificadas.

Os objetos `datetime` também podem ser criados de uma forma semelhante ao objeto `date`. Veja no exemplo:

```
>>> from datetime import datetime
>>> dt = datetime(2014, 1, 1)
>>> dt
datetime.datetime(2014, 1, 1, 0, 0)
```

Ou, podem ser criados com especificação de hora:

```
>>> from datetime import datetime
>>> dt = datetime(2014, 1, 1, 8, 30)
>>> dt18 = datetime(2014, 1, 1, 18, 30)
>>> dt8 = datetime(2014, 1, 1, 8, 30)
>>> dt18 > dt8
True
```

Na prática, quando queremos converter datas, a partir de texto, em objetos do tipo `date` ou `datetime`, usamos o método `strptime()` do objeto `datetime`. Essa pequena mudança ajuda a apagar algumas linhas de código. Veja o exemplo a seguir:

```
>>> from datetime import datetime
>>> datetime.strptime('01/01/2014', '%d/%m/%Y')
datetime.datetime(2014, 1, 1, 0, 0)
```

Da mesma forma, a operação inversa pode ser feita com o método `strftime()`, como no exemplo seguinte:

```
>>> from datetime import datetime
>>> dt = datetime(2014, 1, 1, 19, 15)
>>> dt.strftime('%Y-%d-%m %H:%M')
'2014-01-01 19:15'
```

A lista com as opções de configurações de formatos de data, usadas por `strptime` e `strftime`, está em https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior.

Outra operação comum é a aritmética com datas. Muitas vezes, queremos somar dias, ou meses, ou qualquer outra unidade compatível a uma data. Em outras situações, queremos saber a diferença em dias ou meses, ou até mesmo horas entre duas datas ou dois horários. Todas essas operações podem ser realizadas com os operadores aritméticos, porém com o detalhe de que elas retornem ou usem objetos do tipo `timedelta`. O objeto `timedelta` representa uma diferença entre duas datas, ou data/hora.

No nosso aplicativo, podemos listar quanto se gastou em execuções financeiras, cujo tempo de vigência foi menor que 11 dias:

```
from decimal import Decimal
from datetime import datetime

total = Decimal('0')
start_date = datetime(2014, 1, 1)

def get_value(info):
    try:
        start_str_date = info[8]
        end_str_date = info[9]
        start_date =
            datetime.strptime(start_str_date, '%d/%m/%Y')
        end_date = datetime.strptime(end_str_date, '%d/%m/%Y')
        date_diff = end_date - start_date
        if date_diff.days < 10:
            str_value = info[5]
            value = Decimal(str_value)
```

```
            return value
    except Exception as e:
        print(info)
        pass
    return Decimal('0')

with open('data/data/ExecucaoFinanceira.csv', 'r') as data:
    for line in data:
        total += get_value(line.strip().split(';'))

print("Total gasto com contrados de menos de 11 dias {}"
    .format(total))
```

Vamos examinar melhor esse objeto que está na variável `date_diff`.

```
>>> from datetime import datetime
>>> start_date = datetime.strptime('01/01/2014', '%d/%m/%Y')
>>> end_date = datetime.strptime('10/01/2014', '%d/%m/%Y')
>>> diff = end_date - start_date
>>> type(diff)
<class 'datetime.timedelta'>
>>> diff.days
9
>>> diff.total_seconds() / 60 / 60 / 24 # segundos/minutos/dias
9.0
```

O módulo `datetime` possui uma classe chamada `timedelta`. Essa classe representa justamente uma diferença entre datas ou horas. Quando realizamos operações aritméticas com datas, o objeto retornado é desse tipo. Com ele, podemos pegar a diferença em número de dias, ou até mesmo em segundos totais. O mesmo tipo de operação, subtração, também pode ser feito com horas. Veja no exemplo:

```
>>> start_time = datetime.strptime('18:00', '%H:%M')
>>> end_time = datetime.strptime('21:30', '%H:%M')
>>> diff = end_time - start_time
>>> diff.seconds / 60 / 60
3.5
```

Vemos que a diferença entre `21:30` e `18:00` é de 3 horas e meia. Também podemos ver o valor em segundos, por meio do atributo `seconds` da classe `timedelta`. Um detalhe importante é não confundir o método `total_seconds()` – que pega o total de segundos entre duas datas quaisquer –, com o atributo `seconds` – que se refere apenas à parte das horas.

Uma outra facilidade da classe `timedelta` é que ela também pode ser facilmente construída à mão, para que seja somada a uma data, retornando uma nova data. Veja um exemplo básico primeiro:

```
>>> birthday = datetime(2014, 1, 1)
>>> next_birthday = birthday + timedelta(days=365)
>>> next_birthday
datetime.datetime(2015, 1, 1, 0, 0)
```

No exemplo prático, podemos tirar uma consulta bem interessante: a quantidade de execuções financeiras assinadas em cada ano. Para isso, criamos datas com a data inicial de cada ano, somamos 365 dias e ficamos com duas datas. Depois, filtramos as execuções financeiras assinadas nesse intervalo e contamos a quantidade encontrada. Veja que, nesse exemplo, usamos quase tudo o que foi visto: `datetime`, `timedelta`, operação de comparação e operação aritmética.

```
from decimal import Decimal
from datetime import datetime, timedelta

totals = {2010:0, 2011:0, 2012:0, 2013:0, 2014:0, 2015:0}

def check_signature_interval(info, year_start_date,
                             year_end_date):
    try:
        start_str_date = info[8]
        end_str_date = info[9]
        start_date = datetime.strptime(start_str_date,
                                       '%d/%m/%Y')
        end_date = datetime.strptime(end_str_date, '%d/%m/%Y')

        if start_date > year_start_date and start_date <
        year_end_date:
```

```
            return 1
    except Exception as e:
        pass

    return 0

for year in totals.keys():
    start_date = datetime(year, 1, 1)
    end_date = start_date + timedelta(days=365)
    with open('data/data/ExecucaoFinanceira.csv', 'r') as data:
        for line in data:
            totals[year] += check_signature_interval(
                line.strip().split(';'), start_date, end_date)

for year, signed in totals.items():
    print("{} execuções assinadas em {}".format(signed, year))
```

A saída:

```
35 execuções assinadas em 2010
91 execuções assinadas em 2011
253 execuções assinadas em 2012
4468 execuções assinadas em 2013
506 execuções assinadas em 2014
0 execuções assinadas em 2015
```

Além disso, temos as funções de obter a data/hora e a data atual, sendo, respectivamente, o método `now()` da classe `datetime`, e o método `today()` da classe `date`. Veja o exemplo:

```
>>> from datetime import datetime, date
>>> datetime.now()
datetime.datetime(2014, 8, 18, 23, 25, 3, 297527)
>>> date.today()
datetime.date(2014, 8, 18)
```

11.3 Conjuntos sem repetição: set()

Outra estrutura de dados muito utilizada é o conjunto, que, em Python, é a estrutura `set`.

Essa estrutura não permite elementos duplicados e nem assegura nenhum tipo de ordem entre elementos, nem índices. Portanto, não é uma sequência como uma lista. Esse objeto permite que algumas operações da Teoria dos Conjuntos [11] – como interseção, união, diferença, diferença simétrica, teste de presença de elemento ou elementos – sejam realizadas em um conjunto, ou conjuntos.

No nosso programa, podemos criar conjuntos de empresas associados a execuções financeiras de valores, em algum intervalo arbitrário, e depois consultar o número distinto de empresas em cada grupo. Por exemplo, podemos criar um conjunto com as empresas associadas a execuções superiores a 1 bilhão, 500 milhões, 100 milhões, 10 milhões, 1 milhão e menos.

Vamos continuar com nosso exemplo:

```
# coding: utf-8

from decimal import Decimal

def get_id_and_value(info, lower):
    value = Decimal(info[5])
    if value > lower:
        return info[2], value
    return None, Decimal(0)

all_companies = set()
intervals = [(Decimal('1000000000'), set()),
             (Decimal('500000000'), set()),
             (Decimal('100000000'), set()),
             (Decimal('10000000'), set()),
             (Decimal('1000000'), set()),
             (Decimal('100000'), set()),
             (Decimal('10000'), set()),
             (Decimal('1000'), set())]

for lower, companies in intervals:
    data = open('data/data/ExecucaoFinanceira.csv', 'r')
    for line in data:
        company_id, contract_value = get_id_and_value(
            line.strip().split(';'), lower)
```

```
        if company_id and not company_id in all_companies:
            companies.add(company_id)
        all_companies.add(company_id)
    data.close()

for lower, companies in intervals:
    print("{} empresas receberam mais de {}".format(
        len(companies), lower))
print("{} empresas no total".format(len(all_companies)))
```

Leia com calma para entender as diversas ideias empregadas nesse pequeno programa.

Primeiro, criamos um conjunto onde colocaremos os identificadores de todas as empresas. Com ele, vamos verificar que uma empresa que recebeu 1 milhão não seja contada novamente quando o valor de teste for um valor menor, como 10 mil.

Depois, criamos uma lista de tuplas, na qual cada uma tem um valor de comparação e um conjunto onde colocaremos os identificadores encontrados.

O loop simplesmente varre todos os registros com um valor mínimo corrente, e inclui no conjunto o identificador da empresa que recebeu mais que o valor mínimo corrente. Sem um conjunto, teríamos uma lista com repetições, já que uma mesma empresa pode ter recebido mais de uma execução financeira. Porém, felizmente, temos uma forma elegante de eliminar a repetição sem ter de introduzir manualmente um mecanismo de eliminação de repetições.

A estrutura de conjunto não se limita apenas a eliminar repetições e a implementar algumas operações matemáticas de conjuntos. Veja alguns exemplos a seguir:

```
>>> set([1, 2, 3, 4, 5]) & set([2, 4])
{2, 4}
>>>
>>> set([1, 2, 3, 4, 5]) | set([6, 7])
{1, 2, 3, 4, 5, 6, 7}
>>>
>>> set([1, 2, 3, 4, 5]).isdisjoint(set([6, 7]))
True
```

```
>>>
>>> set([1, 2, 3, 4, 5]) > set([2, 4])
True
>>>
>>> set([1, 2]) < set([1, 2, 3, 4, 5])
True
>>>
>>> set([1, 2, 3]) - set([1, 3])
{2}
>>>
>>> set([1, 2, 3]) ^ set([2, 3])
{1}
```

As operações são:

- `&` – interseção;
- `|` – união;
- `original.isdisjoint(other)` – se o conjunto `other` é disjunto do conjunto `original`;
- `set1 > set2` – se `set2` é subconjunto de `set1`;
- `set1 < set2` – se `set1` é subconjunto de `set2`;
- `-` – diferença entre conjuntos;
- `^` – diferença simétrica que retorna os elementos que estão em apenas um dos conjuntos.

Vale lembrar que, no Python 3.x – posteriormente portado para a família 2.7 –, existe a sintaxe literal de conjuntos, que é como se fosse uma lista ou uma tupla, mas com chaves (como em `{2, 4, 6}`, o conjunto com os números 2, 4 e 6). A sintaxe literal de conjuntos suporta até mesmo o formato de *comprehension*, que veremos mais à frente e que será detalhado melhor quando vermos recursos mais avançados da linguagem, no capítulo 13.

11.4 Tuplas nomeadas: tuplas que parecem objetos com atributos

Muitas linguagens não possuem tuplas e, para quem está tendo contato com elas pela primeira vez, elas podem parecer menos úteis do que realmente são. Um dos seus usos comuns é algo semelhante a `struct` do C, se pensarmos nela como um *container* de outros objetos.

O problema aparente é que as tuplas "comuns" exigem acesso via notação de índice:

```
>>> execucao = ('1', '2', '132', '-1', '76',
...             '696648486.09', '0', '19/03/2010',
...             '23/03/2010', '05/10/2013')
>>> value = execucao[5]
>>> value
'696648486.09'
```

As tuplas nomeadas são subclasses das tuplas, que parecem instâncias de classes de usuário, mas não são. Internamente, tuplas nomeadas e classes de usuário são muito diferentes. O que as faz se parecerem com as classes de usuário é que podemos definir um nome, e um nome para cada atributo que queremos definir na tupla nomeada.

A vantagem da tupla nomeada sobre uma instância de uma classe é que a tupla é um objeto menor, em termos de espaço em memória. Se considerarmos uma quantidade grande de objetos, pode ser mais vantajoso tê-los representados em tuplas nomeadas do que em instâncias de classes "regulares".

É claro que podemos usar tuplas normais sempre, porém, a facilidade de acessar atributos na sintaxe, semelhante ao acesso de atributos de uma instância de classe, é muito conveniente.

Veja como poderíamos utilizá-la:

```
from collections import namedtuple

ExecucaoFianceira = namedtuple('ExecucaoFianceira',
                               ['IdExecucaoFinanceira',
                               'IdEmpreendimento',
```

```
                                'IdInstituicaoContratado',
                                'IdPessoaFisicaContratado',
                                'IdLicitacao',
                                'ValContrato',
                                'ValTotal',
                                'DatAssinatura',
                                'DatInicioVigencia',
                                'DatFinalVigencia'])

execucao = ExecucaoFianceira(
    '1',
    '2',
    '132',
    '-1',
    '76',
    '696648486.09',
    '0',
    '19/03/2010',
    '23/03/2010',
    '05/10/2013'
)

print(execucao.ValContrato)
print(execucao)
```

Na criação da `namedtuple`, primeiro definimos seu nome e, depois, em uma lista, definimos os nomes dos seus atributos. Mais tarde, podemos criar instâncias dessa tupla nomeada e referenciar os atributos pelos seus próprios nomes em vez de sua posição.

11.5 Conclusão

Neste capítulo, vimos decimais, datas, data/hora, hora, conjuntos e tuplas nomeadas. Python possui mais tipos especiais disponíveis, porém estes acabam sendo os mais comuns. Vimos como eles podem nos ajudar com problemas do dia a dia e a melhorar. Na maioria dos casos, é sempre melhor procurar se já não existe algo que precisamos, pois usar as implementações oficiais pode nos poupar de mais uma "reinvenção da roda".

Se você quiser ver mais algum tipo especial explicado aqui na próxima edição, entre em contato pelo site http://livropython.com.br/. Vale também lembrar da nossa lista de discussão, em livro-python3@googlegroups.com.

No próximo capítulo, veremos alguns conceitos e padrões adotados na linguagem, como objetos iteráveis, closures e outros.

Capítulo 12

Conceitos e padrões da linguagem

Existem alguns conceitos e padrões no universo Python que são muito comuns de serem usados ou citados em documentações, tutoriais e outros materiais. Até então, focamos nos aspectos mais básicos e fundamentais da linguagem.

Agora, vamos ver alguns conceitos um pouco mais avançados e que explicam determinados comportamentos ou funcionalidades da linguagem. A terminologia também é importante, pois torna a comunicação com outros *pythonistas* mais fácil, já que muitos estão (ou estarão) familiarizados com algumas coisas que veremos agora.

12.1 Iteráveis e por que o comando for funciona?

Já vimos como realizar loops em coleções – como listas ou dicionários –, usando o comando `for`, mas, no material do livro, não foi explicado por qual razão ele funciona. Esse comando pode ser usado com diversos tipos de objetos, desde que eles sejam *iteráveis*. Diversos objetos da linguagem são iteráveis, como listas, strings, tuplas etc. Essa característica foi incorporada pela PEP-234 [5], que introduziu o conceito de interface de iteração.

Mas o que significa um objeto ser iterável? Para que isso serve?

Iterável é qualquer conjunto – seja lista, dicionário ou tupla –, que pode ter seus elementos visitados um a um, em uma ordem qualquer.

Na prática, uma grande vantagem é que podemos plugar qualquer objeto dentro do `for`, inclusive os que nós criamos, desde que o tornemos um `iterável`. Um objeto, para ser considerado iterável, tem de implementar o método `__iter__()`. Geralmente, a implementação retorna um objeto chamado de iterador (*iterator* em inglês), que, caso implemente ambos os métodos `__iter__()` e `__next__()`, pode ser plugado no comando `for`.

Podemos pensar que um objeto que representa uma consulta, ou uma *query*, pode ser um iterável. Mas, como conseguimos isso?

Para tal, vamos criar um objeto iterável que lê, registro a registro, o arquivo de dados e nunca cria um objeto com todos eles em memória, simultaneamente. Se esse objeto for um iterador, podemos usá-lo no comando `for`.

Vamos ver como esse objeto seria:

```
# coding: utf-8

from decimal import Decimal

total = Decimal('0')

def dec(element, index):
    try:
        return Decimal(element[index])
    except:
        return Decimal('0')
```

```
class QueryFile():
    def __init__(self, filename):
        self._file = open(filename, "r")

    def __iter__(self):
        return self

    def __next__(self):
        data = self._file.readline()
        if not data:
            self._file.close()
            raise StopIteration
        return data.split(';')


query = QueryFile('data/data/ExecucaoFinanceira.csv')
total = sum(dec(element, 5) for element in query)

print("Total gasto: {}".format(total))
```

Repare que eliminamos completamente o uso de listas. O objeto do tipo `QueryFile` é o objeto *iterador* que usamos para percorrer todos os registros do arquivo de dados. Ele que mantém a referência para o arquivo aberto, que, por sua vez, mantém atualizado o ponteiro das próximas leituras de dados e avança, linha a linha, usando a função `readline()`, que lê apenas uma linha quando é chamada.

Outro detalhe importante é que o final do percurso é sinalizado com a geração de `StopIteration`. Podemos dizer que nossa classe `QueryFile` implementa o protocolo de *iteráveis* (*iterator protocol*), portanto, pode ser usada com o comando `for`.

12.2 Objetos chamáveis: callables()

Um outro termo extremamente utilizado no universo Python é *callable*. Um objeto que pode ser invocado com a sintaxe de chamada, usando parênteses, é um *callable*. Os maiores exemplos de callables são as funções e os métodos.

O método construtor, por exemplo, funciona, pois em Python as classes são *callables*, e o ato de chamar o construtor pela classe dispara a chamada do método `__init__()`, definido por ela.

O grande detalhe é que é possível tornar qualquer classe um callable. Basta que esta implemente o método especial `__call__()`. Podemos modificar o exemplo anterior, fornecendo uma interface de consulta por meio da sintaxe de chamada, sem utilizar um método, mas sim, um mecanismo de chamada.

Veja o exemplo:

```
# coding: utf-8
from decimal import Decimal

class QueryFile():
    def __init__(self, filename):
        self._file = open(filename, "r")

    def __iter__(self):
        return self

    def __next__(self):
        data = self._file.readline().split(';')
        if not data or len(data) == 1:
            self._file.close()
            raise StopIteration

        returned_el = []
        for position in self.positions:
            if len(data) >= position:
                returned_el.append(data[position])

        return returned_el

    def __call__(self, *args):
        self.positions = args
        return self
```

```
query = QueryFile('data/data/ExecucaoFinanceira.csv')

for data in query(5, 7):
    print("Execucao no valor de {} assinada {}".format(
        data[0], data[1]))
```

Repare que, na chamada à `query()`, passamos os índices dos atributos que queremos ver. Com a informação dos metadados, podemos trocá-los para receber as strings que identificam os campos, como descrito no arquivo de metadados.

12.3 Protocolos abstratos: como comportamentos padronizados funcionam

Os protocolos abstratos são uma outra característica muito importante da linguagem Python. Esses protocolos definem conjuntos de comportamentos que padronizam o uso de alguns objetos. O exemplo mais comum e que já foi mencionado no livro é o protocolo de *sequência*.

Se analisarmos, o uso de listas e strings é muito semelhante em muitos aspectos. Em ambas, podemos acessar elementos por um índice ou *slice*, testar a presença de um elemento nelas e perguntar o tamanho. Veja os exemplos a seguir, para relembrar:

```
>>> st = "gastos publicos"
>>> len(st)
15
>>> st[0]
'g'
>>> st[-1]
's'
>>> "g" in st
True
>>> gastos = [1000.0, 2000.0, 3000.0]
>>> gastos[0]
1000.0
>>> gastos[-1]
3000.0
```

```
>>> 1000.0 in gastos
True
```

Mesmo sendo objetos completamente diferentes, sua manipulação é igual para essas operações. Isso porque ambos implementam o protocolo de *sequência*. Até mesmo o comportamento dos objetos iteráveis é definido por um protocolo abstrato, que exige a implementação dos métodos `__iter__()` e `__next__()`.

Os dicionários implementam o protocolo abstrato de mapas (em inglês, *mapping*). Os mapas também permitem a pergunta pelo seu tamanho, o pedido de um valor dada uma chave e a iteração na tupla chave/valor. É relativamente comum encontrarmos materiais na internet que afirmam que tal objeto é *"dict-like*

. Isso significa que podemos manipulá-lo e que ele não é necessariamente um dicionário – como se fosse um –, já que provavelmente implementa o protocolo abstrato de mapas.

Veremos mais concretamente os protocolos abstratos no capítulo 14.

12.4 Closures em Python

Em Python, temos escopo léxico e *funções de primeira classe*. Com esses elementos, podemos usar uma técnica chamada de *closure*.

A técnica consiste em retornar uma função que use internamente variáveis (ou nomes) da função que a define.

Imagine que queremos uma função que retorne `True`, caso um valor seja maior que `100000`.

Poderíamos escrevê-la da seguinte forma:

```
>>> def greater_100K(val):
...     return val > 100000
...
>>>
>>> greater_100K(99999)
False
>>> greater_100K(99999 + 2)
True
```

O problema do código anterior é que ele está amarrado ao número 100000, dentro da função do código. Com *closures*, podemos criar algo mais elegante e flexível, que permita que as funções sejam criadas em tempo de execução, de acordo com os valores escolhidos. Veja o exemplo:

```
>>> def greater(fixed_val):
...     def _greater(val):
...         return val > fixed_val
...     return _greater
...
>>> greater_100K = greater(100000)
>>> greater_100K(99999)
False
>>> greater_100K(99999 + 2)
True
```

Esse tipo de "técnica" é muito comum em programação funcional e em Python, já que esta é uma linguagem multiparadigma, o que torna isso possível.

Mesmo não sendo algo exclusivo de Python, pessoas que conhecem outras linguagens podem nunca ter tido contato com closures. É importante saber o que são e como usá-las em Python.

12.5 Conclusão

O objetivo deste capítulo foi transmitir uma parte da terminologia, que se manifesta em características da linguagem, técnicas de programação, padrões de projeto e de outras formas.

Uma linguagem de muito alto nível como Python pode, em um primeiro olhar, tornar tudo tão simples que os detalhes passam sem serem notados. Em um segundo momento, alguns desenvolvedores se questionarão mais sobre como as coisas funcionam. Nesta segunda parte do livro, continuaremos vendo aspectos da implementação e design da linguagem, o que eles permitem e como usá-los.

Capítulo 13
Elementos com sintaxe específica

Agora, veremos alguns elementos específicos de linguagem, que talvez não sejam exclusivos, mas certamente são marcantes no universo Python. Vamos mudar um pouco a abordagem do livro. Em vez de iniciar as explicações com as motivações práticas, vamos ver gradativamente o novo elemento apresentado e, ao final de cada seção, um exemplo aplicado ao contexto do nosso programa.

13.1 List comprehensions

Uma variação na sintaxe que causa certa estranheza, principalmente para quem vem de linguagens com sintaxes mais tradicionais – como C ou Java –, é a *list comprehension*. Uma tarefa muito comum em programas é criar listas a partir de outras listas, ou dentro de loops. Isso pode ser feito com o comando `for` que já vimos, mas a *list comprehension* fornece uma forma mais elegante,

em muitos casos, de se fazer o mesmo. No início do livro, usamos o comando `for` para imprimir os números de 0 até 4:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Esse código pode ser facilmente adaptado para que uma lista com os números impressos seja criada:

```
>>> numbers = []
>>> for i in range(5):
...     numbers.append(i)
...
>>> numbers
[0, 1, 2, 3, 4]
```

A sintaxe de *list comprehension* mistura uma sintaxe de declaração de lista com o comando `for` dentro dela.

```
>>> numbers = [i for i in range(5)]
>>> numbers
[0, 1, 2, 3, 4]
```

Aqui, vemos que o bloco que seria executado para cada valor de `i` virou uma expressão, que é retornada o número de vezes que a expressão do `for` – no caso, `range(5)` – retornar um elemento. Dessa forma, conseguimos criar essa lista de números com apenas uma linha. Além disso, os elementos retornados podem ser manipulados, e até condicionais podem ser colocadas no meio:

```
>>> numbers_times_2 = [i*2 for i in range(5)]
>>> numbers_times_2
[0, 2, 4, 6, 8]
```

```
>>> evens = [i for i in range(5) if i % 2 == 0]
>>> evens
[0, 2, 4]
```

Indo mais além, da mesma forma que podemos aninhar comandos `for`, podemos aninhar *list comprehensions*. Veja o exemplo usando o comando `for`, sem a sintaxe nova:

```
>>> nested = []
>>> for i in range(2):
...     for j in range(3):
...         nested.append((i, j))
...
>>> nested
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Agora, o mesmo código usando *list comprehension:*

```
>>> nested = [(i, j) for i in range(2) for j in range(3)]
>>> nested
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

No primeiro exemplo do capítulo 12 (seção 12.1), criamos um código que soma o valor de todas as execuções financeiras. Podemos tornar aquele código mais elegante com o emprego de *list comprehensions*.

```
# coding: utf-8

from decimal import Decimal

total = Decimal('0')

def dec(element, index):
    try:
        return Decimal(element[index])
    except:
        return Decimal('0')

with open('data/data/ExecucaoFinanceira.csv', 'r') as data:
    splited_data = [line.split(';') for line in data]
```

```
    total = sum([dec(element, 5) for element in splited_data])

print("Total gasto: {}".format(total))
```

Aplicamos *list comprehensions* duas vezes. A primeira foi para montar uma lista de listas. A variável `splited_data` é uma lista de registros do arquivo de execuções financeiras, e cada elemento desta é uma lista com os valores descritos no arquivo de metadados para tabela.

Se ficou um pouco confuso, veja como deve ficar a estrutura do objeto `splited_data`:

```
splited_data = [
    [1, 'DOC001', ... , 'Decimal('12345.00')'],
    [2, 'DOC002', ... , 'Decimal('12345.00')'],
    [3, 'DOC003', ... , 'Decimal('12345.00')'],
    [4, 'DOC004', ... , 'Decimal('12345.00')']
]
```

Note que construímos, com apenas uma linha, uma lista de listas. Dessa estrutura, nós montamos uma outra lista, apenas com a coluna de índice 5, que é a coluna com o valor que queremos.

A função `sum()` recebe algo como `[Decimal('123.00'), Decimal('321.00')]`, e soma todos os elementos. O detalhe importante é que, a partir do arquivo, montamos uma lista com todos os dados, cujo tamanho está amarrado ao número de registros. Ou seja, se temos 100 mil elementos, vamos ter uma lista com 100 mil elementos, todos simultaneamente em memória.

Talvez isso não seja um problema em um programa que calcula a soma e logo termina, porém, em processos de longa duração e com vários clientes, como em sistemas web, não queremos ficar criando listas gigantescas na memória. Na seção 13.5, sobre *funções geradoras*, veremos como contornar essa situação.

13.2 Dicts comprehensions

De forma análoga à *list comprehensions*, a *dict comprehension* funciona da mesma forma, porém o resultado é um dicionário. No exemplo, vamos ver a

função `zip()`, que é bem comum em programas Python para criar dicionários a partir de listas.

Imagine que queremos que cada resultado seja um dicionário, em que a chave é o nome do atributo e o valor é o próprio valor desse atributo. Por exemplo, podemos ter um objeto como `{'NumDocumento': '2014NE800370', 'ValContrato': 'Decimal('13499.20')'}`. Veja o exemplo a seguir:

```
# coding: utf-8

from decimal import Decimal

total = Decimal('0')
schema = ('NumDocumento', 'ValContrato')

def dec(element, index):
    try:
        return Decimal(element[index])
    except:
        return Decimal('0')

with open('ExecucaoFinanceira.csv', 'r') as data_file:
    splited_data = [line.split(';') for line in data_file]
    data = [(element[2], dec(element, 12)) for element in
        splited_data if len(element) > 2]
    result = [{key:value for key, value in zip(schema, element)}
        for element in data]

    for info_dict in result:
        print("{}".format(info_dict))
```

Repare que os elementos de `result` são os dicionários criados com a *dict comprehension*. A função `zip()` também nos ajudou, permitindo que a tupla `schema`, com a coleção de atributos, e a lista de dados que queremos sejam iteradas simultaneamente, de modo que seja possível criar um dicionário que junte essas duas listas. O resultado desse programa seria algo como:

```
...
{'NumDocumento': '"2014NE800934"',
```

```
 'ValContrato': Decimal('445.10')}
{'NumDocumento': '"2014NE800316"',
 'ValContrato': Decimal('368.62')}
{'NumDocumento': '"2014NE801005"',
 'ValContrato': Decimal('38950.00')}
{'NumDocumento': '"2014NE800317"',
 'ValContrato': Decimal('95.89')}
{'NumDocumento': '"2014NE800318"',
 'ValContrato': Decimal('222.16')}
...
```

Talvez a *dict comprehension* não seja tão comum quanto sua versão em lista, mas seu emprego é relativamente comum e, quando usada na situação mais adequada, pode deixar o código mais elegante e sucinto.

13.3 Decoradores de função: @decorators

No capítulo 3, vimos que, em Python, as funções são objetos como todos os outros, comumente chamadas de *first class objects*. Por isso, podemos criar funções que recebem outras como parâmetro, e retornam funções também. Especialmente, essa combinação de receber uma função e retornar outra é uma das formas como um *decorator* pode ser implementado.

Na prática, o *decorator* é um mecanismo de sintaxe que chama uma função, passando outra como parâmetro, ou chama o construtor de uma classe, passando uma função como parâmetro. Pensando em alto nível, um *decorator* é uma informação declarativa sobre uma função em questão. Ele nos permite adicionar comportamentos extras a funções existentes, de forma declarativa. Isso é um recurso que pertence à metaprogramação.

Exemplo prático do uso de um decorator

Vimos um uso prático no capítulo 7, quando usamos o decorator `@property`. Esse decorador é um encapsulador do acesso a atributos de um objeto, isto é, ele permite que sejam definidas funções que serão chamadas no caso de três operações com objetos:

- Acesso, como em `table.name`;

- Atribuição, como em `table.data = []` (veja que vale qualquer tipo de atribuição válida);
- Remoção, como em `del table.data`.

Em Python, não existem variáveis privadas, mas existe a convenção de se usar um *underscore* (_) antes do nome das variáveis, para passar esse significado. Para que todo o resto do programa não use essa variável privada diretamente, podemos definir uma propriedade que a representa. Para ela, podemos associar funções que serão executadas nos três eventos descritos anteriormente. Vamos ver um exemplo muito simples:

```
>>> class Column:
...     def __init__(self, name):
...         self._name = name
...     @property
...     def name(self):
...         return self._name
...     def other_name(self):
...         return self._name
...
>>> col = Column('Name')
>>> col.other_name
<bound method Column.other_name of <__main__.Column object at
0x1053a6588>>
>>> col.name
'Name'
```

O objetivo do decorator `@property` é criar um atributo artificial chamado `name`, que representa o acesso de leitura à variável `_name`. Repare que a implementação de `name` e de `other_name` são iguais, mas como `other_name` não está com o decorator `@property`, a referência para ela, sem o uso da sintaxe de chamada de função com `()`, faz com que uma instância do método seja retornada em vez de o método ser invocado. Com o decorator `@property`, a referência a `col.name` invoca o método `name` e, por isso, o retorno é o valor de `_name`.

De forma semelhante, podemos definir um encapsulador para atribuição de valores, para um atributo de um objeto. Por exemplo, quando criamos uma

coluna, não queremos que o valor do seu nome mude em tempo de execução. Para isso, devemos impedir que novos valores sejam atribuídos. Isso pode ser feito com um *setter*. Veja o exemplo:

```
>>> class Column:
...     def __init__(self, name):
...         self._name = name
...     @property
...     def name(self):
...         return self._name
...     @name.setter
...     def name(self, value):
...         raise Exception("Nome não pode mudar")
...     @name.deleter
...     def name(self):
...         pass
...
>>>
>>> col = Column('Name')
>>> col.name = 'New Name'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in name
Exception: Nome não pode mudar
```

No dia a dia, podemos ver decoradores que exploram apenas o fato de serem chamados quando a função é chamada, mas não alteram o resultado da chamada original, e outros que o fazem, ou alteram a própria chamada original.

A seguir, vamos ver em detalhes como o mecanismo funciona, para que fiquem mais claros todos os recursos disponíveis que ganhamos ao utilizar *decorators*.

Como funciona um decorator?

Em Python, podemos implementar um *decorator* de duas formas. A forma mais comum é definindo-o como uma função; a outra – menos comum, porém igualmente poderosa – é usando uma classe com métodos es-

peciais. Primeiro, vamos ver em mais detalhes a primeira forma, e depois veremos a segunda.

Antes de pensar no decorator, pense em uma função que recebe uma função como parâmetro.

```
>>> def uma_funcao():
...     print("uma_funcao")
...
>>> def outra_funcao(func):
...     print("outra_funcao")
...     func()
...
>>> outra_funcao(uma_funcao)
outra_funcao
uma_funcao
```

Como vimos na seção 3.11, as funções são objetos de primeira classe, portanto podem ser passadas como parâmetros para outras funções. Até aí, nenhuma novidade.

Essa característica também permite que uma função seja retornada por outra. Veja o exemplo:

```
>>> def uma_funcao_com_args(param):
...     print("uma_funcao param {}".format(param))
...
>>> def outra_funcao(func, *args):
...     print("Irei chamar {}".format(func.__name__))
...     return func(*args)
...
>>> outra_funcao(uma_funcao_com_args, [1, 2, 3])
Irei chamar uma_funcao_com_args
uma_funcao param [1, 2, 3]
```

Repare que usamos `outra_funcao()` apenas para escrever qual função vamos chamar. Veja que, como temos o recurso do *packing*, conseguimos usar esse recurso para um caso genérico como:

```
>>> outra_funcao(sum, [1, 2, 3])
Irei chamar sum
```

```
6
>>> outra_funcao(max, [1, 2, 3])
Irei chamar max
3
```

Além de conseguir executar um código arbitrário (no caso, apenas um print), conseguimos executar a função passada como parâmetro, com o restante dos parâmetros.

A função dos decoradores é adicionar, de forma declarativa, um comportamento extra a uma função. Essa facilidade é muito útil e elegante.

O exemplo real que veremos logo a seguir mostra um *decorator* que imprime na saída padrão o nome da função chamada e os argumentos passados para ela. Toda função que estiver com esse nosso decorador ganhará esse novo comportamento.

Esse seria o exemplo mais básico de um decorator. Veja o exemplo:

```
def trace_call(func):
    def inner(*args, **kwargs):
        print("Function executed: {} args: {}".format(
                                    func.__name__, args))
        func(*args, **kwargs)
    return inner

@trace_call
def add(x, y):
    return x + y

add(5, 1)
```

Gera a saída:

```
Function executed: add args: (5, 1)
```

Primeiro, veja que `trace_call()` é a função decoradora. A função que chamamos de original, nesse exemplo, é `add()`, e a função retornada é `inner()`.

Agora, vamos entender todos os detalhes. A função `trace_call` é um decorator que foi implementado como função. Dentro dela, definimos uma

outra função, chamada `inner`, que, quando chamada, imprime uma informação na tela e chama a função original `add()`, representada pela variável `func`, com os parâmetros originais.

Além disso, a função decoradora `trace_call` tem de retornar uma referência para `inner`. Assim, quando chamamos `add()`, internamente a função chamada é `trace_call()`, que recebe a função `add()` como parâmetro e retorna `inner`. Esta executa uma tarefa qualquer e, depois, executa `add()`.

Dessa forma, toda função decorada com `@trace_call`, antes de ser executada, tem seu nome e seus argumentos impressos.

Quando definimos um decorator, podemos inclusive adicionar parâmetros na chamada original, mudar parâmetros, ou qualquer outra coisa que seja compatível. O maior ponto de atenção é que os decorators precisam ter um objetivo bem definido e realmente tornar o programa mais reusável e claro.

Para implementar um *decorator* usando uma classe, você deve criar uma classe que implementa o método especial `__call__()`. Veja o decorador `trace_call` reimplementado como classe:

```
class trace_call:
    def __init__(self, f):
        self.f = f
    def __call__(self, *args, **kwargs):
        print("Function executed: {} args: {}".format(
                                        self.f.__name__, args))
        self.f(*args, **kwargs)

@trace_call
def add(a, b):
    return a + b

add(1, 3)
```

Esse programa gera a saída:

```
Function executed: add args: (1, 3)
```

13.4 Funções anônimas: lambdas

As funções anônimas são empregadas, geralmente, quando uma determinada função for usada somente uma vez, ou poucas vezes. A ideia de função anônima é amplamente difundida nas linguagens funcionais. Python, como linguagem multiparadigma, tem a sua versão, que são chamadas de *lambdas*.

Em Python, os lambdas não podem conter comandos (*statements*), decoradores e nem múltiplas linhas, mas podem conter várias expressões. Um emprego prático muito comum é seu uso com a função da biblioteca padrão, `filter()`.

A assinatura completa é `filter(function, iterable)`, que chama a função passada para cada elemento e, quando esta retorna `True`, a função `filter` retorna esse elemento.

No nosso aplicativo, podemos passar um lambda como critério de um filtro, que seleciona valores maiores que um determinado número:

```
# coding: utf-8
from decimal import Decimal

total = Decimal('0')
_100M = Decimal('100000000')

def dec(element, index):
    try:
        return Decimal(element[index])
    except:
        return Decimal('0')

with open('ExecucaoFinanceira.csv', 'r') as data:
    splited_data = [line.split(';') for line in data]
    values = [dec(element, 12) for element in splited_data]
    total = sum(values)
    values_100M = filter(lambda x: x > _100M, values)
    total_gt_100M = sum(values_100M)

percent = lambda x, y: (x/y) * Decimal('100')

print("Total gasto: {}".format(total))
```

```
print("Apenas contratos com mais de 100MI: {}".format(
                                          total_gt_100M))
print("Representam {:.2f}% do total".format(percent(
                                    total_gt_100M, total)))
```

No exemplo anterior, empregamos lambdas duas vezes: primeiro, passando uma função anônima para função `filter`, para selecionar apenas valores maiores que 100 milhões; depois, atribuímos um lambda a uma variável – o que é válido –, e a função deixa de ser anônima.

Isso não é um problema, e é relativamente comum encontrar atribuições de funções anônimas a variáveis, quando essas funções não fazem nenhuma declaração ou executam algum comando dentro dela.

Já houve tentativas de alterar algumas coisas em lambdas, como a possibilidade de usar várias linhas na sua definição, mas até agora todas fracassaram.

13.5 Funções geradoras

Uma característica muito relevante na linguagem é a existência de funções geradoras. Esse tipo de função especial permite algo que pode ser implementado com listas, ou com um iterador, mas em ambos os casos existem problemas.

Imagine que queremos somar o total de execuções financeiras, da forma como fizemos, aplicando a função `sum()`, em uma lista montada com uma *list comprehension*. Quando a lista for muito grande – como, por exemplo, com mais de 1 milhão de elementos –, já devemos pensar duas vezes se queremos criar um objeto desse tamanho em memória.

No caso de um iterador, temos um pequeno problema: é necessário um pouco de codificação para criá-lo. Felizmente, a própria linguagem já possui uma forma natural para termos algo que funciona como um iterador, mas que não exige a criação de coleções em memória.

Na prática, a função geradora permite que ela retorne um elemento e, quando for chamada novamente, consiga restaurar o estado de execução anterior e retornar um novo elemento. Veja o exemplo que compara uma função que gera os números de 1 a 10, usando uma lista, e um exemplo com uma função geradora.

```
>>> def get_list():
...     numbers = []
...     for i in range(10):
...         numbers.append(i)
...     return numbers
...
>>> for number in get_list():
...     print(number)
...
0
1
2
3
4
5
6
7
8
9
>>> def get_generator():
...     for i in range(10):
...         yield i
...
>>> for number in get_generator():
...     print(number)
...
0
1
2
3
4
5
6
7
8
9
```

Sempre que usamos a palavra reservada `yield`, estamos criando uma função geradora. No exemplo, fica claro que ela é capaz do mesmo efeito que

a função que cria uma lista, porém ela não usa uma lista. A função geradora "gera" um valor toda vez que é "chamada" e, aqui, usamos aspas, pois não é uma chamada comum. Por padrão, as funções geradoras podem ser iteradas no comando `for`, que já sabe como obter os novos valores gerados por ela.

No nosso programa isso é muito útil, pois, para percorrermos todos os dados dos arquivos, não precisamos criar nenhuma lista com eles em memória. Basta criarmos uma função que, a cada linha, gera um valor para a função chamadora.

Vamos ver um código em específico e explicá-lo logo em seguida:

```
>>> my_generator = get_generator()
>>> my_generator
<generator object get_generator at 0x1034d6120>
```

Repare que `my_generator` é um objeto do tipo `generator`. Isso significa que é uma função geradora. Vamos tentar obter o primeiro valor, no caso 0 (zero).

```
>>> my_generator()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not callable
```

Uma função geradora não é chamável (*callable*), portanto temos esse erro. Para obter os valores gerados, usamos a função `next(generator_function)`. Esta é que faz com que a função geradora seja executada, gere um elemento e tenha seu estado preservado, até que outra chamada com `next()` seja feita. Veja o exemplo:

```
>>> my_generator = get_generator()
>>> next(my_generator)
0
>>> next(my_generator)
1
>>> next(my_generator)
2
>>> next(my_generator)
3
```

```
>>> next(my_generator)
4
>>> next(my_generator)
5
... # outras execuções
>>> next(my_generator)
9
>>> next(my_generator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Repare que, quando os elementos terminam, temos uma `StopIteration`, que é justamente a exceção que temos que levantar no iterador quando não houver mais elementos para retornar. Isso porque as funções geradoras são iteráveis (*iterables*).

Na prática, as funções geradoras são muito usadas porque resolvem o problema de criar objetos que podem ocupar muito espaço, de uma forma bem simples, usando apenas uma palavra reservada.

Além disso, existem as expressões geradores (em inglês, *generator expressions*), que têm sintaxe semelhante às *list comprehensions*, mas não criam listas em memória.

```
>>> sum(x for x in [10]*10)
100
```

13.6 Palavra reservada nonlocal

Quando falamos de *closures* no capítulo 12 (seção 12.4), vimos que uma função pode retornar outra, e a função retornada pode usar variáveis do escopo da função que a define. Veja outro exemplo:

```
>>> def make_counter(count):
...     def counter():
...         return count
...     return counter
...
>>> count = make_counter(0)
```

```
>>> count()
0
>>> count()
0
>>> count()
0
```

Aqui, a função interna usa uma variável no escopo da que a define. Se você rodar esse código, verá como resultado 3 zeros.

Mas o que queremos é modificar o valor do contador a cada chamada da função `count()`. Nesse caso, podemos tentar alterar o valor de `count` dentro da função interna. Veja o exemplo e o resultado a seguir:

```
>>> def make_counter(count):
...     def counter():
...         count += 1
...         return count
...     return counter
...
>>> count = make_counter(0)
>>> count()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in counter
UnboundLocalError: local variable 'count' referenced before
assignment
```

A razão disso é, apesar de podermos ler, não podemos alterar o valor de `count` dentro de `counter()`. Com a PEP-3104 [15], foi introduzido um mecanismo formal para tornar isso possível. A palavra reservada `nonlocal` informa que estamos usando uma variável (ou nome), definida em um escopo imediato externo. Com o seu uso, torna-se possível alterar o valor de `count`, para termos um contador de verdade.

Veja o exemplo:

```
>>> def make_counter(count):
...     def counter():
...         nonlocal count
```

```
...             count += 1
...             return count
...         return counter
...
>>> count = make_counter(0)
>>> count()
1
>>> count()
2
>>> count()
3
```

Nas versões anteriores, conseguíamos algo semelhante ao usar objetos mutáveis, como uma lista. Eles podem ser alterados sem o uso da palavra `nonlocal`. Veja o exemplo:

```
>>> def make_counter():
...     acc = []
...     def counter(val):
...         acc.append(val)
...         return acc
...     return counter
...
>>> count = make_counter()
>>> count(1)
[1]
>>> count(2)
[1, 2]
>>> count(3)
[1, 2, 3]
```

No fundo, a PEP-3104 buscou dar mais clareza à linguagem, oferecendo uma maneira explícita e abrangente de usarmos o recurso de alterar variáveis definidas em escopos externos.

13.7 Conclusão

Neste capítulo, vimos diversos recursos que utilizam uma sintaxe específica. Além disso, vimos o propósito de cada um deles e algumas noções de como

utilizá-los no seu dia a dia. Esses recursos são extremamente comuns em códigos Python que encontraremos por aí. Logo, é importante dominá-los e, principalmente, aplicá-los quando for a melhor opção.

Mais à frente, veremos alguns assuntos mais avançados de classes, e explicaremos diversos fatores que não foram mostrados no primeiro material de classes. Vamos ver como o modelo de objetos Python integra-se com alguns aspectos da sintaxe da linguagem.

Capítulo 14

Explorando a flexibilidade do modelo de objetos

O modelo de classes de Python é simples, porém muito flexível. Neste capítulo, vamos além do que aprendemos no capítulo 7, para aprender outras tarefas igualmente importantes quando estamos definindo classes para os nossos programas. Ao final do capítulo, saberemos como criar classes, com todos os comportamentos mais comuns implementados, e ver quais são e como eles funcionam.

Vamos ver também alguns métodos especiais e mecanismos importantes no sistema de objetos Python. Ao final, já estaremos aptos para explorar e obter mais facilidades do sistema de objetos.

14.1 Construtures: criando objetos

Na primeira apresentação de classes, no capítulo 7, vimos que, para definir o método construtor, basta implementar o método `__init__()` na nossa classe. Esse método especial é chamado quando fazemos uma invocação no objeto da classe. A ausência de construtor faz com que, automaticamente, a classe tenha um construtor sem parâmetros.

```
>>> class DataTable:
...     pass
...
>>> table = DataTable()
```

Caso seja definido um construtor que recebe um parâmetro, já não é mais possível instanciar sem passá-lo:

```
>>> class DataTable:
...     def __init__(self, name):
...         self._name = name
...
>>> table = DataTable('ExecucaoFinanceira')
>>> table = DataTable()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument:
'name'
```

Em Python, caso sejam declarados mais de um método construtor, o último a ser declarado acaba sendo o que vale. Veja o exemplo:

```
>>> class DataTable:
...     def __init__(self, name):
...         self._name = name
...     def __init__(self, data):
...         self._data = data
...
>>> t = DataTable('ExecucaoFinanceira')
>>> t._name
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AttributeError: 'DataTable' object has no attribute '_name'
>>> t._data
'ExecucaoFinanceira'
```

Em Java, existe a possibilidade de termos diversos construtores com assinaturas diferentes. Entretanto, em Python, geralmente isso se resolve com apenas um construtor e parâmetros com valor padronizado. Veja o exemplo a seguir:

```
>>> class DataTable:
...     def __init__(self, name, data=[]):
...         self._name = name
...         self._data = data
...
>>> t = DataTable('ExecucaoFinanceira')
>>> t._data
[]
>>> t = DataTable('ExecucaoFinanceira',
                  data=['linha1', 'linha2'])
>>> t._data
['linha1', 'linha2']
>>> t._name
'ExecucaoFinanceira'
```

Repare que oferecemos duas formas de criar um objeto `DataTable`, mas usando apenas um construtor e um parâmetro com valor padronizado.

O último detalhe em relação ao método construtor é quando precisamos invocar o construtor da superclasse no construtor da subclasse. Vamos lembrar da classe `PrimaryKey`, que herda de `Column`.

```
class PrimaryKey(Column):
    def __init__(self, table, name, kind, description=""):
        Column.__init__(self, name, kind,
                        description=description)
        self._is_pk = True
```

A chamada ao construtor da superclasse tem um detalhe muito importante: ela chama o método `__init__` direto na superclasse, e passa a sua instância subclasse, como primeiro argumento do construtor da superclasse.

```
>>> class Column:
...     def __init__(self, name, kind, description=""):
...         print(type(self))
...
>>> class PrimaryKey(Column):
...     def __init__(self, table, name, kind, description=""):
...         Column.__init__(self, name, kind,
                            description=description)
...         self._is_pk = True
...
...     def get_name(self):
...         return self._name
...
>>> pk = PrimaryKey(None, 'PK', 'int')
<class '__main__.PrimaryKey'>
```

Repare que o tipo impresso é `PrimaryKey`. Isso porque passamos uma instância de `PrimaryKey`, diretamente para o parâmetro `self` do construtor da superclasse (no caso, `Column`). Essa passagem é feita pela linha `Column.__init__()`.

O último detalhe é que construtores não podem retornar explicitamente nenhum valor. Caso retorne, você ganhará um `TypeError`, com uma mensagem semelhante a `TypeError: __init__() should return None, not ...`.

14.2 Representação textual de objetos: função print(), str() e repr()

Outra funcionalidade comum (e esperada em linguagens com suporte à Orientação a Objetos) é como definir a representação textual de uma classe. Muitas vezes queremos imprimir, no console, o conteúdo de uma classe de uma forma estruturada, sem ter que ficar imprimindo todos os seus atributos, um a um.

Se voltarmos às classes definidas anteriormente, no capítulo 7, e usarmos uma instância delas na função `print()`, veremos uma representação padronizada que, muitas vezes, não é o que queremos ver. Vamos rever um código

simplificado desse capítulo:

```
>>> class DataTable:
...     def __init__(self, name):
...         self._name = name
...
>>> table = DataTable("ExecucaoFinanceira")
>>> print(table)
<__main__.DataTable object at 0x109277860>
```

Veja que o resultado da chamada da função retorna um texto genérico. Ele exibe o módulo `__main__` – porque foi definido no console –, o nome da classe e uma representação do endereço onde o objeto se encontra.

Assim como em outras linguagens – como Java ou C++ –, é possível customizar a representação em texto de nossas classes. Em cada linguagem isso é feito de uma forma específica e, em Python, fazemos isso implementando um de dois (ou os dois) possíveis métodos especiais: `__str__()` ou `__repr__()`.

Vamos ver um exemplo usando `__str__()`:

```
class DataTable:
    def __init__(self, name):
        self._name = name
    def __str__(self):
        return "Tabela {}".format(self._name)

>>> table = DataTable("ExecucaoFinanceira")
>>> print(table)
Tabela ExecucaoFinanceira
>>> table
<__main__.DataTable object at 0x109277b00>
>>> str(table)
'Tabela ExecucaoFinanceira'
```

De forma semelhante, podemos definir o método `__repr__()`:

```
>>> class DataTable:
...     def __init__(self, name):
...         self._name = name
```

```
...     def __repr__(self):
...         return "Tabela {}".format(self._name)
...
>>> table = DataTable("ExecucaoFinanceira")
>>> print(table)
Tabela ExecucaoFinanceira
>>> table
Tabela ExecucaoFinanceira
>>> str(table)
'Tabela ExecucaoFinanceira'
```

Repr versus Str

A documentação define que a representação retornada pelo método especial `__str__()` é uma representação "informal", usada apenas quando o objeto é passado como parâmetro para a função `str(object)`, ou quando passado para a função `print()`. Repare que, no primeiro exemplo, na segunda exibição do objeto continuamos vendo sua representação genérica.

O método `__repr__()` deve retornar uma representação "formal", que deve idealmente permitir que o objeto seja recriado a partir da sua representação em texto. Se o objeto não implementar `__str__()`, mas implementar `__repr__()`, ele será usado quando uma representação "informal" for solicitada. No console, quando a entrada for apenas o nome de uma variável, a sua representação "oficial" é chamada, e o método `__repr__()` é invocado.

Esse detalhe de `__str__()` *versus* `__repr__()` já foi muito discutido na internet. O que importa é que seja usado o que for mais conveniente para o seu programa.

14.3 Comparando igualdade de objetos

Outra operação comum com objetos é a comparação de igualdade. Em Python, como tudo é um objeto, se compararmos as instâncias das classes que criamos até o momento, o resultado será sempre `False`, pois nenhuma delas implementa o método especial de comparação. Isso ocorre porque o mecanismo de comparação padrão compara os identificadores dos objetos, que são únicos para cada instância, mesmo que elas contenham exatamente

os mesmos valores.

Como eu poderia saber se duas tabelas são a mesma?

Para termos uma comparação de igualdade que leve em consideração os valores dos atributos do objeto, temos que implementar o método especial `__eq__()`. Este é muito semelhante ao método `equals()` de Java, ou `Equals()` de C#.

Sua assinatura deve ter um parâmetro, que é o objeto com o qual a instância está sendo comparada. Dentro desse método, podemos, então, fazer uma comparação de atributos que seja compatível com nosso domínio. Por exemplo, no nosso aplicativo, tabelas com mesmo nome são iguais.

Veja o exemplo **sem** a implementação o método `__eq__()`:

```
>>> class DataTable:
...     def __init__(self, name):
...         self._name = name
...
>>> t1 = DataTable("ExecucaoFinanceira")
>>> t2 = DataTable("ExecucaoFinanceira")
>>> t1 == t2
False
>>> id(t1)
4510145448
>>> id(t2)
4510145392
```

Repare que o resultado da comparação foi `False`, porque, embora tenham mesmo nome, são instâncias com identificadores diferentes. Quando realizamos uma comparação de igualdade sem o método `__eq__()` definido, os identificadores são comparados.

Vamos agora ao exemplo em que o método `__eq__()` é implementado:

```
>>> class DataTable:
...     def __init__(self, name):
...         self._name = name
...     @property
...     def name(self):
...         return self._name
```

```
...     def __eq__(self, other):
...         return self.name == other.name
...
>>> t1 = DataTable("ExecucaoFinanceira")
>>> t2 = DataTable("ExecucaoFinanceira")
>>> t1 == t2
True
>>> id(t1)
4510145952
>>> id(t2)
4510146008
```

Agora, mesmo tendo identificadores diferentes, a comparação retornou `True`. Internamente, a comparação `t1 == t2` invoca o método `__eq__()` em `t1`, passando `t2` como parâmetro.

Sabendo disso, basta colocar a lógica de comparação dentro desse método. No nosso caso, as tabelas que têm o mesmo nome são iguais. É claro que, dentro desse método, você deve fazer as comparações que forem necessárias, para que seu domínio esteja bem representado no código.

Método cmp() removido na versão 3.x

Em muitas versões da família 2, a comparação era feita por meio do método especial `__cmp__`. O protocolo que esse método deveria obedecer implicava em "forçar" uma ordem artificial em objetos que, semanticamente, podem não ter relação de ordem. O problema é que, pelo protocolo, ou um é maior que o outro, ou eles são iguais e fica impossível representar a ausência de ordem, sem dizer que eles são iguais.

Mais tarde, isso foi considerado ruim. Então, foi decidido que seriam criados um método por operador de comparação, como ==, > e outros. Na versão 3.x temos o `__eq__()` e outros que veremos na sequência.

14.4 Outros tipos de comparações binárias

Já sabemos que a comparação com `var1 == var2` invoca o método `__eq__()` em `var1`, passando como parâmetro `var2`. E isso vale para

outras operações também. Para compactar a informação, vamos ver a tabela a seguir, que mapeia os métodos especiais aos comparadores:

```
object.__lt__(self, other) | x < y
object.__le__(self, other) | x <= y
object.__ne__(self, other) | x != y
object.__gt__(self, other) | x > y
object.__ge__(self, other) | x >= y
```

Para cada método implementado, ganhamos a opção de usar a expressão correspondente. Dessa forma, se implementarmos `__lt__()`, poderemos usar `var1 < var2`. Vamos ver um exemplo:

```
>>> class TesteLT:
...     def __init__(self, n):
...         self.n = n
...     def __lt__(self, other):
...         return self.n < other.n
...
>>> TesteLT(3) < TesteLT(4)
True
>>> TesteLT(4) < TesteLT(3)
False
```

Esse exemplo ilustra bem o emprego de `__lt__()`. Não é extremamente comum implementar algum desses métodos, porém, é importante saber que eles existem e como funcionam.

14.5 Dicionário interno de atributos

Como já mencionado no início do livro, Python implementa várias de suas funcionalidades usando suas estruturas de dados. *A priori*, todos os objetos têm seus atributos guardados em um dicionário interno, chamado de `__dict__`. Esse objeto é chamado de dicionário interno de atributos.

```
>>> class DataTable:
...     def __init__(self, name, data=[]):
...         self._name = name
```

```
...         self._data = data
...     def add_row(self, row):
...         self._data.append(row)
...
>>> t = DataTable('ExecucaoFinanceira')
>>> t.__dict__
{'_data': [], '_name': 'ExecucaoFinanceira'}
```

Repare que `t.__dict__` é um dicionário normal, cujas chaves/valores são, respectivamente, nome e valor dos atributos da instância criada. Se você sentiu falta dos atributos referentes aos métodos, veja o exemplo a seguir:

```
>>> DataTable.__dict__.keys()
dict_keys(['__module__', '__dict__', '__doc__', '__init__',
'__weakref__', 'add_row'])
```

Repare que os métodos estão definidos no objeto da classe, e não nas instâncias. Por isso, os dicionários `t.__dict__` e `DataTable.__dict__` são diferentes.

Esse recurso é muito poderoso, mas é apenas para casos muito específicos. Frameworks ou ferramentas que fazem algum tipo de *patching* do código, em alguns casos, o fazem através da manipulação direta do dicionário interno de atributos. Isso pode ser feito por meio de algumas funções embutidas, como: `setattr()`, `getattr()` e `hasattr()`.

Veja o exemplo do uso de `setattr()`:

```
>>> class DataTable:
...     pass
...
>>> t = DataTable()
>>> setattr(t, 'name', 'ExecucaoFinanceira')
>>> t.name
'ExecucaoFinanceira'
```

Note que definimos um atributo na instância, ao usarmos a função `setattr(object, attribute_name, value)`. Essencialmente, teríamos o mesmo resultado com `t.name = 'ExecucaoFinanceira'`, mas a

vantagem de `setattr()` é que podemos passar uma string arbitrária, como nome do atributo.

A operação de obter o valor do atributo, geralmente feita com `objeto.atributo`, pode também ser feita com a função `getattr(object, attribute_name)`. Veja o exemplo a seguir:

```
>>> class DataTable:
...     pass
...
>>> t = DataTable()
>>> setattr(t, 'name', 'ExecucaoFinanceira')
>>> t.name
'ExecucaoFinanceira'
>>> getattr(t, 'name')
'ExecucaoFinanceira'
```

Essencialmente, `t.name` é o mesmo que `getattr(t, 'name')`. A diferença é que, no segundo caso, também podemos passar uma string arbitrária.

Por último, vemos o método `hasattr(obj, attribute_name)`, que responde se determinado objeto tem algum atributo com o nome `attribute_name`.

```
>>> class DataTable:
...     def __init__(self, name, data=[]):
...         self.name = name
...         self._data = data
...     def add_row(self, row):
...         self._data.append(row)
...
>>> t = DataTable('ExecucaoFinanceira')
>>> hasattr(t, 'name')
True
>>> hasattr(t, '_data')
True
```

Essa flexibilidade é muito interessante, mas, assim como no caso do *ducktyping*, vale sempre pensar bem se existe a necessidade real de se usar esses recursos.

14.6 Interceptando o acesso a atributos

Por meio de métodos especiais, podemos também definir uma forma de acesso customizado aos atributos de um objeto. A ideia é interceptar a chamada que acessa um atributo, para executar um código arbitrário.

Em Python, quando usamos uma expressão como `table.name`, no fundo, estamos realizando `getattr(table, 'name')`, ou seja, queremos acessar o atributo `name` do objeto `table`. Por padrão, esse acesso é feito no dicionário interno de atributos, que vimos anteriormente.

Em alguns casos, pode ser interessante poder modificar como essa busca de atributo é feita. Um caso famoso dessa necessidade é em modelos de plugins, no qual os atributos só aparecem em tempo de execução, e fica impossível criar todos os métodos/atributos, sem saber previamente quais os códigos o cliente pode querer usar.

Também existe um padrão que pode explorar bem essa característica, que é o *Proxy*. Neste, temos um objeto que representa outro e, de certa forma, é uma "camada" entre o cliente e o objeto que está atrás do proxy. Algumas coisas de programação orientada a aspectos também podem ser realizadas com a implementação desses métodos.

Veja um exemplo:

```
>>> class Proxy:
...     def __init__(self, obj):
...         self.obj = obj
...
...     def __getattr__(self, name):
...         print("Acesso ao atributo {}".format(name))
...         if hasattr(self.obj, name):
...             return getattr(self.obj, name)
...         else:
...             raise Exception("Atributo desconhecido")
...
>>> class DataTabte:
...     def __init__(self, name):
...         self.name = name
...
>>> class Query:
```

```
...     def __init__(self, attributes):
...         self.attributes = attributes
...
>>> table_proxy = Proxy(DataTable('ExecucaoFinanceira'))
>>> query_proxy = Proxy(Query(['id', 'valor']))
>>> table_proxy.__dict__
{'obj': <__main__.DataTable object at 0x100a48e48>}
>>> query_proxy.__dict__
{'obj': <__main__.Query object at 0x100a48c88>}
>>> print(table_proxy.name)
Acesso ao atributo name
ExecucaoFinanceira
>>> print(query_proxy.attributes)
Acesso ao atributo attributes
['id', 'valor']
```

Por meio da implementação do método `__getattr__()`, conseguimos fazer com que o uso de uma instância do nosso `Proxy` seja muito semelhante ao uso do próprio objeto encapsulado dentro dele. Repare que os atributos que acessamos não existem nos dicionários internos do proxy, mas, quando usamos, acessamos os atributos, e o proxy busca no objeto encapsulado e retorna.

De forma semelhante, temos o método `__setattr__()`, que é acionado quando atribuímos um valor a um atributo de um objeto, como por exemplo: `table.nome = "ExecucaoFinanceira"`. Isso tem o mesmo resultado que `setattr(obj, "nome", "ExecucaoFinanceira")`. Vamos ver um exemplo ilustrando seu uso:

```
>>> table = DataTable()
>>> table.name = "ExecucaoFinanceira"
>>>
>>> table.name
'ExecucaoFinanceira'
>>> class DataTable:
...     pass
...
>>> table = DataTable()
>>> table.name = "ExecucaoFinanceira"
```

```
>>>
>>> class DataTable:
...     def __setattr__(self, name, value):
...         raise Exception("Classe de leitura apenas")
...
>>> table = DataTable()
>>> table.name = "ExecucaoFinanceira"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in __setattr__
Exception: Classe de leitura apenas
```

No caso do `__setattr__()` e do `__getattr__()`, eles são executados quando não existe definição para o atributo em questão. Por exemplo, quando acessamos um atributo que não foi definido, tanto para obter seu valor quanto uma atribuição, um dos métodos será executado.

Caso você precise que os métodos sejam executados em todos acessos a atributos, você pode definir os métodos `__getattribute__()` e `__setattribute__()`. Veja um exemplo que ilustra bem isso:

```
>>> class DataTable:
...     def __init__(self, name):
...         self._name = name
...     def __getattr__(self, attr_name):
...         print("Attributo não definido '{}' acessado".format(
                                                    attr_name))
...         if attr_name == "data":
...             return []
...         raise AttributeError("Atributo '{}' não existe"
                                             .format(attr_name))
...     def __getattribute__(self, attr_name):
...         print("Attributo {} acessado".format(attr_name))
...         return object.__getattribute__(self, attr_name)
...
>>> t = DataTable("ExecucaoFinanceira")
>>> t._name
Attributo _name acessado
'ExecucaoFinanceira'
>>> t.data
```

```
Attributo data acessado
Attributo não definido 'data' acessado
[]
>>> t.cols
Attributo cols acessado
Attributo não definido 'cols' acessado
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: Atributo 'cols' não existe
```

Ao tentarmos acessar o atributo `_name`, quando ele existe, apenas `__getattribute__()` é acessado. Ele repassa a execução para `__getattribute__()` da classe `object`, que vai buscar o atributo no dicionário interno da nossa classe.

Ao acessarmos o atributo `data`, quando ele não existe, primeiro o `__getattribute__` é acessado. A implementação do método `__getattribute__()`, por sua vez, repassa a execução para a implementação desse mesmo método, só que na classe `object`.

A implementação de `__getattribute__()` em `object` procura o atributo pelo nome no dicionário interno. Como ele não foi definido, o método `__getattr__()` é chamado. Dentro deste, retornamos uma lista vazia, caso o nome do atributo seja `'data'`, ou geramos uma exceção de `AttributeError`. Esta é normalmente usada nesse tipo de situação, e foi ilustrada quando tentamos acessar o atributo `cols` do objeto.

14.7 Protocolos e métodos especiais

Como vimos no capítulo 12, Python tem protocolos, usados para definir comportamentos padronizados, para determinados tipos de objetos. Para implementar um protocolo, um conjunto específico de métodos especiais deve ser realizado.

Diversas estruturas de dados seguem protocolos, assim como números também têm seus protocolos. Os protocolos, inclusive, influenciam na terminologia usada pela comunidade. Da próxima vez que você ouvir algo como "esse objeto é um iterável" ou "essa função funciona com qualquer objeto com-

patível com uma sequência", já saberá que essas frases se referem a protocolos e têm definição formal.

Do ponto de vista de um autor de biblioteca, os protocolos podem ser uma forma de criar APIs mais integradas aos recursos da linguagem. Isso permite um uso mais "leve" e menos orientado, em chamadas de métodos em objetos.

Um exemplo completo é o protocolo de sequências. Já comentamos sobre ele anteriormente em [?], mas não vimos explicitamente que métodos fazem parte dele.

Uma sequência é um objeto que: responde se um outro objeto pertence a ele, responde ao seu tamanho, retorna um objeto por índice e é iterável. Esse protocolo é composto de outros protocolos menores, e os comportamentos mais complexos acabam emergindo da união de comportamentos menores definidos.

Falando em protocolos mais básicos, vamos ver os mais fundamentais das estruturas de dados, em Python.

14.8 Protocolo de container

Um container, em Python, tem de implementar apenas um comportamento: responder se um elemento qualquer faz parte ou não do container. Fica fácil ver que listas, tuplas, dicionários, strings e outras estruturas são containers. O método especial é `__contains__()`.

```
>>> class TenhoTudo:
...     def __contains__(self, obj):
...         return True
...
>>> tem_tudo = TenhoTudo()
>>> 1 in tem_tudo
True
>>> "qualquercoisa" in tem_tudo
True
>>> [] in tem_tudo
True
```

Como você pode ver, sempre retornamos `True`. Logo, o operador `in` sempre retornará esse valor quando usado em um objeto `TemTudo`.

No contexto do nosso aplicativo, esse método poderia ser usado para responder se determinado objeto encontra-se em determinada coluna de informação.

Por exemplo, se uma tabela contém uma determinada chave, o container responde `True`:

```
>>> table = DataTable("ExecucaoFinanceira",
            "execucaoFinanceira.csv")
>>> 1 in table
True
```

Veja que, por meio da implementação de `__contains__()`, nosso `DataTable` pode responder se determinada chave primária está entre seus elementos. Apesar de ser um exemplo simples, se construirmos uma API que implementa diversos protocolos, ela acaba tornando-se bem amigável ao uso e bem integrada ao resto da linguagem.

14.9 Protocolo de containers com tamanho definido

O nome real do protocolo que vamos ver agora é o *sized protocol*, também chamado de "container com tamanho definido". Esse protocolo também tem apenas um método, o `__len__()`. Basta o objeto sempre saber responder o seu tamanho, que pode ser considerado um *sized*.

Ele também pode ser implementado na nossa `DataTable` para responder, na verdade, o tamanho de linhas de informação que ela contém. Veja um exemplo ilustrativo:

```
>>> class DataTable:
...     def __init__(self, name, filename):
...         self._name = name
...         with open(filename) as data:
...             self._data = data.readlines()
...     @property
...     def name(self):
...         return self._name
...     def __len__(self):
```

```
...         return len(self._data)
...
>>> table = DataTable("ExecucaoFinanceira",
                      "execucaoFinanceira.csv")
>>> len(table)
5375
```

Novamente, usamos o protocolo a nosso favor, para tornar a API um pouco menos complexa, pois não exigimos que o usuário saiba qual atributo contém os dados para chamar `len()` e obter o número de registros.

14.10 Protocolo de iteráveis e iteradores

Outro protocolo fundamental e muito importante é o de objetos iteráveis. Todo objeto que o implementa pode ser usado no comando `for` e em *comprehensions*. Quando informamos que um objeto é iterável, significa que podemos percorrer os seus elementos de forma sequencial.

No nosso exemplo, podemos tornar um `DataTable` iterável, e os elementos iterados seriam os seus registros. Para ser compatível com esse protocolo, a única exigência é que o método `__iter__()` seja implementado e retorne um objeto iterador. O objeto iterador é o objeto que representa, em si, o fluxo de dados.

O objeto **iterador**, por sua vez, tem de implementar o método `__next__()`, que deve retornar o próximo elemento do percorrimento. É dentro de `__next__()` que se controla a ordem em que os objetos são retornados.

Em muitos casos, o objeto iterável também é um iterador dele mesmo. Por isso, é comum que eles implementem `__iter__()` e `__next__()`.

Vamos tornar nosso objeto iterável e iterador:

```
>>> class DataTable:
...     def __init__(self, name, filename):
...         self._name = name
...         self._c = 0
...         with open(filename) as data:
...             self._data = data.readlines()
```

```
...     @property
...     def name(self):
...         return self._name
...     def __len__(self):
...         return len(self._data)
...     def __iter__(self):
...         return self
...     def __next__(self):
...         try:
...             element = self._data[self._c]
...         except IndexError as ie:
...             self._c = 0
...             raise StopIteration
...         self._c += 1
...         return element
...
>>> table = DataTable("ExecucaoFinanceira",
                      "execucaoFinanceira.csv")
>>> for line in table:
...     print(line)
...
37145;354;11982;xxxxxxxxxxxxxxxxxx;
14332.00;0;25/07/2013;25/07/2013;31/12/2013

37147;354;11831;xxxxxxxxxxxxxxxxxx;
832.88;0;26/07/2013;26/07/2013;31/12/2013

37148;360;12581;xxxxxxxxxxxxxxxxxx;
590.60;0;04/06/2014;04/06/2014;14/06/2014

37149;354;12899;xxxxxxxxxxxxxxxxxx;
1200.00;0;26/07/2013;26/07/2013;31/12/2013
```

Veja que, no exemplo, implementamos ambos os métodos. Dessa forma, podemos usar o objeto `DataTable` direto no comando `for`, mais uma vez, integrando melhor nossa API com a linguagem.

A implementação é bem simples: basicamente, mantemos um índice do elemento corrente a ser retornado, no caso, a variável "privada" `_c`. A função `__next__()` sempre retorna o elemento da lista `_data`, apontado por esse

índice. Quando chegamos ao final, levantamos `StopIteration` para sinalizar que o loop iniciado terminou. Nesse momento, reiniciamos o contador para 0, para que o objeto possa ser percorrido do início em uma próxima vez.

14.11 Protocolo de sequências

Se implementarmos os três protocolos anteriores e mais um método, vamos atender ao protocolo de **sequência**.

No fim das contas, para ser uma sequência, o objeto deve ser um container com tamanho definido, iterável e que é capaz de retornar um elemento, dado um índice. Esse último comportamento se dá por meio do método `__getitem__()`.

Em sequências, esse método deve receber um inteiro ou *slice object*. Qualquer outro tipo deve ser interpretado como erro, preferencialmente gerando um `TypeError`. Se o argumento for um inteiro, deve ser retornado o objeto que ocupa a posição representada pelo inteiro. Se for um *slice object*, devem ser retornados os elementos correspondentes a esse *slice*.

No nosso caso, a implementação de `__getitem__()` simplesmente retorna o objeto da sua lista interna de registros.

```
>>> class DataTable:
...     def __init__(self, name, filename):
...         self._name = name
...         self._c = 0
...         with open(filename) as data:
...             self._data = data.readlines()
...     @property
...     def name(self):
...         return self._name
...     def __len__(self):
...         return len(self._data)
...     def __iter__(self):
...         return self
...     def __next__(self):
...         try:
...             element = self._data[self._c]
...         except IndexError as ie:
```

```
...                 self._c = 0
...                 raise StopIteration
...             self._c += 1
...             return element
...         def __getitem__(self, i):
...             if isinstance(i, int) or isinstance(i, slice):
...                 return self._data[i]
...             raise TypeError("Invalid index/slice object '{}'"
                                .format(str(i)))
...
>>> table = DataTable("ExecucaoFinanceira",
                      "execucaoFinanceira.csv")
>>> print(table[0])
1;2;132;CONSTRUTORA ANDRADE GUTIERREZ S/A.;696648486.09;0;
19/03/2010;23/03/2010;05/10/2013
>>> print(table[1:3])
['2;3;357;MENDES JUNIOR TRADING E ENGENHARIA S. A.;
342060007.96;0;
20/04/2010;20/04/2010;03/06/2013\n',
'5;6;136;Arena das Dunas Concess\xc3\xa3o e Eventos S.A.;
400000000.00;0;31/10/2011;31/10/2011;31/10/2031\n']
>>> print(table['1'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 25, in __getitem__
TypeError: Invalid index/slice object '1'
```

Tudo que fizemos foi repassar o objeto recebido para a lista interna de registros. Como essa lista é uma sequência, ela é compatível com os objetos do tipo inteiro ou *slice*. Caso o objeto passado não seja nenhum dos dois, levantamos um erro do tipo `TypeError`.

Com a implementação dos métodos `__contains__()`, `__len__()`, `__iter__()`, `__next__()` e `__getitem__()`, tornamos o nosso objeto `DataTable` compatível com o protocolo de sequência. Qualquer função que espera um objeto compatível com esse protocolo pode receber um objeto desse tipo. Além disso, integramos o nosso objeto a diversos comandos da linguagem, como o operador binário `in`, o comando de loop `for`, a função `len()` e a sintaxe de acesso a elementos `objeto[indice]`.

Além de *sequence*, temos a definição de uma *mutable sequence*, que deve implementar métodos que permitem a mudança de elementos. Para ser uma sequência mutável, o objeto deve ser uma sequência e implementar, pelo menos, os métodos `__setitem__()`, `__delitem__()` e `insert`.

O método `__setitem__()` implementa o comportamento da atribuição em uma posição, e `__delitem__()` o comportamento de apagar um valor de uma posição. Veja os exemplos:

```
>>> lista = [1, 2, 3, 4, 5]
>>> lista[2] = 10
>>> lista
[1, 2, 10, 4, 5]
>>> del lista[1]
>>> lista
[1, 10, 4, 5]
>>> lista.insert(0, -1)
>>> lista
[-1, 1, 10, 4, 5]
```

Existe uma implementação de sequência mutável `collections.abc.MutableSequence`, que tem alguns métodos concretos que se utilizam dos métodos fundamentais descritos anteriormente. Se sua classe herdar de `MutableSequence`, você ganha os métodos adicionais: `append()`, `reverse()`, `extend()`, `pop()`, `remove()`, e `__iadd__()` (veremos esse método em detalhes na seção 14.13).

Alguns objetos são apenas sequências, como tuplas e strings. Já lista é um objeto que é sequência mutável.

14.12 Protocolo de mapas

Outro protocolo fundamental é o de mapas (`Mapping`). As funções que devem ser implementadas são `__getitem__()`, `__iter__()` e `__len__()`, além dos protocolos de container, container com tamanho definido (`Sized`) e de iteráveis.

O objeto mais famoso que implementa esse protocolo é o dicionário (`dict`). Existe também o mapa mutável (`MutableMapping`), que necessita

dos métodos `__setitem__()` e `__delitem__()`, além dos métodos necessários em `Mapping`.

Geralmente, quando queremos customizar esses métodos, a princípio partimos das classes abstratas do módulo `collections.abc` (https://docs.python.org/3/library/collections.abc.html) , especificamente `Mapping` e `MutableMapping`.

Mesmo que pareça confuso, os métodos `__setitem__()`, `__getitem__()` e `__contains__()` são válidos tanto para *sequências* quanto para *mapas*. A diferença é muito importante!

Nas *sequências*, os objetos são indexados por inteiros (`int`) e, em alguns casos, podem receber *slice objects* como parâmetro (seção 14.11). No caso dos *mapas*, os objetos passados, como em `valor = objeto['atributo']`, podem ser strings ou qualquer objeto imutável que implemente o protocolo `Hashable`, como strings, números ou até mesmo tuplas.

14.13 Protocolos numéricos

Além dos métodos especiais vistos, existem vários outros, com destaque para os que são usados quando operações numéricas são executadas. Expressões – como `a + b` ou `a >> b` – também disparam métodos especiais.

> A referência completa e oficial de métodos está disponível em https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types.

No mundo real, encontramos o uso dos métodos especiais numéricos onde o objeto em questão representa um número – por exemplo, a classe `Decimal`, vista no capítulo 11. Isso nos permite manipular esses objetos em expressões numéricas, como no código `Decimal("1") + Decimal("2.5")`, que resulta em `Decimal("3.5")`.

Em uma linguagem como Java, que não possui esse tipo de recurso, **sempre** temos de invocar um método – como em `BigDecimal result = new BigDecimal(1).add(new BigDecimal(2.5));` –, já que a linguagem

não suporta expressões numéricas com objetos, mas apenas com tipos primitivos e com os respectivos objetos associados a eles (classe `Integer` ou `Long`, por exemplo).

Em alguns casos mais exóticos, autores de projetos dão outras semânticas aos métodos especiais numéricos, para criar APIs mais expressivas. No caso do nosso aplicativo, podemos criar uma sintaxe exótica para realizar uma consulta nos dados. Veja o exemplo:

```
>>> class DataTable:
...     def __init__(self, name, filename):
...         self._name = name
...         with open(filename) as data:
...             self._data = data.readlines()
...     @property
...     def name(self):
...         return self._name
...     def __len__(self):
...         return len(self._data)
...
>>> class Select:
...     def __init__(self):
...         self._c = 0
...     def __lshift__(self, table):
...         self.table = table
...         return self
...     def __iter__(self):
...         return self
...     def __next__(self):
...         try:
...             element = self.table._data[self._c]
...         except IndexError as ie:
...             self._c = 0
...             raise StopIteration
...         self._c += 1
...         return element
...
>>> select = Select()
>>> table = DataTable("ExecucaoFinanceira",
...                   "../data/data/execucaoFinanceira.csv")
```

```
>>>
>>> for line in select << table:
...     print(line)
```

Repare que foi possível usar a expressão `select << table`, por meio da implementação de `__lshift__()`. Esse exemplo foi criado mais pelo fim ilustrativo do que prático. Entretanto, ainda assim, alguns projetos que existem por aí fazem o uso desse tipo de recurso, algo que pode ser mais criativo e útil. O mais importante é conhecer o mecanismo, não só dos métodos especiais numéricos, mas de todos em geral. Podemos esbarrar com eles muitas vezes.

14.14 O que mais ver sobre classes e modelo de objetos?

Ainda existem outros assuntos sobre classes; por serem mais avançados, não fazem parte do escopo deste livro, portanto fica por conta do leitor se aprofundar neles. Um bom desenvolvedor Python deve conhecer também o método especial `__new__()`, responsável pela criação de uma nova instância de um objeto, diferente de `__init__()`, que deve ser responsável pela inicialização.

O atributo especial `__slots__` também pode ser útil quando estivermos manipulando uma grande quantidade de objetos, pois permite que menos memória seja ocupada pelas instâncias deles.

Outro assunto relevante são as metaclasses, que dão ainda mais poder e flexibilidade para metaprogramação, em Python.

14.15 Conclusão

A existência dos protocolos permite que nossos objetos se integrem melhor aos recursos já disponíveis nativamente na linguagem ou na biblioteca padrão. Com eles, podemos manipular objetos criados por nós, com construções e expressões da linguagem, fugindo um pouco de APIs totalmente implementadas com métodos (como no caso de Java, em versões um pouco mais antigas).

A flexibilidade oferecida pela possibilidade de implementarmos protocolos (quando conveniente), torna o modelo de objetos Python muito poderoso e ao mesmo tempo simples. É importante sempre relembrar que recursos desse tipo devem ser usados quando realmente for necessário.

Os recursos explicados aqui são usados por alguns projetos e podemos esbarrar com eles no nosso dia a dia. Mesmo que, em alguns casos, a aplicação não seja tão ampla, é importante saber como as coisas funcionam, pois isso permite melhor uso e entendimento dos recursos disponíveis.

Capítulo 15

Considerações finais

15.1 Nunca pare!

Este livro não é um guia completo sobre Python e nunca pretendeu ser. Veja-o como um ponto de partida.

O universo Python é muito rico e a linguagem, sem sombra de dúvidas, é muito produtiva. Tenha em mente que um maior domínio reflete em uma produtividade ainda maior. É claro que todo aprendizado é gradual.

Neste livro, tentei selecionar alguns tópicos que já apresentam vários temas recorrentes, principalmente quando falamos em iniciantes, tanto em Python quanto em programação em geral.

15.2 Aprendi tudo?

Claro que não! :)

A primeira dica é que programação, independente da linguagem, exige estudo e prática. Algumas pessoas optam por se lançar na prática com menos estudo, e outras fazem ao contrário. De qualquer forma, é importante alternar as duas coisas. Crie pequenos projetos e, preferencialmente, termine-os, mesmo que eles tenham um objetivo bem limitado.

Tente absorver bem a parte da teoria, antes de ir para assuntos mais avançados, para não acumular lacunas de conhecimento. Participe de DOJOS, caso tenha disponibilidade – se não souber o que é, faça uma pesquisa na internet! Não tenha pressa, mas estabeleça metas e tente cumpri-las.

Vamos definir dois tipos principais de usuários, como simplificação do mundo real, para você entender como o livro pode ajudá-lo. Veja com qual deles você se identifica hoje, para então ver como o livro atende às suas necessidades.

Usuários de linguagem

Um usuário de uma linguagem a vê como uma ferramenta para resolver seus problemas. Hoje, temos diversos usuários de Python que resolvem problemas, incrivelmente complexos, usando Python, mas não necessariamente são *experts* da linguagem.

Python tem crescido muito, pois é relativamente fácil de se tornar um usuário da linguagem. Assim, ela torna-se produtiva com certa rapidez.

Um usuário, geralmente, faz muito uso de código de terceiros e usa os aspectos mais fundamentais da linguagem no seu dia a dia. O seu foco de trabalho são seus problemas, e não a linguagem em si.

Desenvolvedor Python

Esse outro "tipo" é alguém com interesse em linguagens de programação ou no ecossistema Python (incluindo outros interpretadores e projetos mais avançados), ou que apenas quer entender precisamente todos os mecanismos e aspectos da linguagem.

Diversos projetos – como debuggers e ferramentas de testes – usam intensivamente conhecimentos bem específicos da linguagem. É muito difícil ser um bom desenvolvedor de alguns tipos de ferramentas sem entender seus detalhes internos e algumas motivações para decisões sobre o seu design.

Mas, e aí?

Se você quer ser um usuário de Python, este livro é um ótimo ponto de partida. Todos os assuntos foram abordados com exemplos e todos eles poderão (e deverão) ser discutidos na internet. Novos exemplos serão criados e mais conteúdo também! Um dos pontos mais fortes de Python é que, sabendo apenas o básico e conhecendo as ferramentas certas, muitos problemas podem ser resolvidos ou estudados.

Se você deseja ser um desenvolvedor Python, o livro será um livro introdutório. Alguns assuntos e detalhes que não são muito comuns em materiais pela internet foram explicados, como também um pouco de teoria. Quando vimos protocolos abstratos e métodos especiais, a importância maior era passar ao leitor a terminologia e o propósito, como ponto de partida para leituras mais profundas, mas não tínhamos como objetivo cobrir individualmente cada um deles.

Existem ainda características como a expressão `yield from`, alguns módulos importantes (como `os`, `sys`, `asyncio`), detalhes a respeito de ASTs, bytecode, máquina virtual e serialização, que são muito relevantes no universo Python, porém exigem mais cuidados e informações para que possam ser passados.

Alguns dos tópicos que são considerados avançados podem ser estudados em material na internet e em outros livros mais avançados, como o *Fluent Python*, escrito por Luciano Ramalho, famoso pythonista brasileiro.

15.3 Comunidade Python

A comunidade Python é **muito** diversificada. Aqui no Brasil temos iniciantes, experts e muitos desenvolvedores de outras linguagens que, de alguma forma, se conectam com a comunidade Python. Além disso, existe um intercâmbio com o meio científico, já que Python é uma linguagem extremamente popular quando se fala em computação científica. Isso tudo torna a comunidade brasileira de Python extremamente rica, que contribui com o amadurecimento das pessoas que se envolvem com ela efetivamente.

Um livro também cria uma comunidade. Todos os exemplos poderão ser discutidos e incrementados, por meio da colaboração de quem quer que seja. Temos um fórum: livro-python3@googlegroups.com e um repositório GitHub oficial, com os exemplos, para que todos possam colaborar com o livro de alguma forma: https://github.com/felipecruz/exemplos.

15.4 Para onde ir?

Primeiro, você deve se perguntar quais são seus objetivos hoje. Qual sua motivação para aprender Python? É a sua primeira linguagem? Você já desenvolve, mas quer uma outra opção como linguagem? Você se beneficiaria no seu trabalho (ou pesquisa) se soubesse programar?

Para cada pergunta, há uma ou várias respostas. Para programadores iniciantes, muito treino é necessário. Mantenha uma disciplina em desenvolver e criar testes quando for importante, e desenvolva algoritmos conhecidos, como *quicksort*, *mergesort*, busca em largura e vários outros, para fixar as operações elementares na linguagem. Pesquise e veja como desenvolvedores mais experientes organizam seus projetos, criam testes, e quais ferramentas usam para aumentar a qualidade ou produtividade.

Caso você já desenvolva, foque-se em aprender como resolver problemas comuns em Python, de preferência da forma *pythônica*. Crie um projeto com script de instalação e testes, e estude como fazer isso da melhor forma. Pesquise ferramentas e frameworks para ir montando seu kit de ferramentas. Converse com outras pessoas! Mas sempre tenha em mente os seus objetivos e não os esqueça! Com bom foco, o ingresso no universo Python pode ser bem rápido e produtivo para quem já é desenvolvedor.

Se você não trabalha com tecnologia, mas está buscando conhecimento sobre programação, saiba que muitos excelentes desenvolvedores são autodidatas, e que você também pode ser um deles. Talvez seja interessante direcionar o estudo da linguagem sempre com um gancho para resolver um problema do seu cotidiano.

Um estatístico pode usar Python como uma ferramenta poderosa para

análises, assim como um economista pode obter grande êxito no seu dia a dia usando scripts simples, mesmo sem se aprofundar mais na linguagem.

Em todos os casos, Python é apenas uma ferramenta que pode ajudá-lo a realizar diversos tipos de tarefa. De qualquer modo, podemos considerar que algumas partes da linguagem são importantes, não importa qual seja o seu caso.

Um resumo de dicas:

1) Estude e pratique!
2) Procure ajuda, mas não seja preguiçoso.
3) Não tente aprender tudo de uma só vez, mas também não fique parado.
4) Participe de DOJOS, eventos de Python.
5) Vá a uma Python Brasil! Tem todo ano!
6) Seja um membro da Associação Brasileira de Python.
7) Participe de comunidades presenciais e virtuais.
8) Ajude pessoas com menos conhecimento, se puder!

Obrigado!

Referências Bibliográficas

[1] Governo Brasileiro. Portal da transparência - copa 2014. http://www.portaldatransparencia.gov.br/copa2014.

[2] Unicode Consortium. Unicode. http://www.unicode.org/.

[3] Python Documentation. Python unlimited integers. http://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex.

[4] Nick Coghlan Guido van Rossum. New import hooks. http://legacy.python.org/dev/peps/pep-0343/, 2005.

[5] Guido van Rossum Ka-Ping Yee. Pep-234 - iterators. http://www.python.org/dev/peps/pep-0234/, 2001.

[6] Martin V Löwis. Pep-393. http://www.python.org/dev/peps/pep-0393/, 2010.

[7] Guido van Rossum Moshe Zadka. Unifying long integers and integers. http://www.python.org/dev/peps/pep-0237/, 2001.

[8] Tim Peters. The zen of python. http://www.python.org/dev/peps/pep-0020/, 2004.

[9] Guido van Rossum. Pep-8. http://www.python.org/dev/peps/pep-0008/, 2001.

[10] Wikipedia. Code points. http://en.wikipedia.org/wiki/Code_point.

[11] Wikipedia. Teoria dos conjuntos. http://pt.wikipedia.org/wiki/Teoria_dos_conjuntos.

[12] Wikpedia. Associative arrays. http://en.wikipedia.org/wiki/Associative_array.

[13] Wikpedia. Ieee floating point. http://en.wikipedia.org/wiki/IEEE_754-2008.

[14] Wikpedia. Utf-8. https://en.wikipedia.org/wiki/UTF-8.

[15] Ka-Ping Yee. Access to names in outer scopes. https://www.python.org/dev/peps/pep-3104/, 2006.